MASTER EN SCIENCES INFORMATIQUES

Contribution à l'étude de l'implémentation d'un langage de microprogrammation de haut-niveau

Dupriez, Jean-Pierre

Award date:
1976

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
A NAMUR
INSTITUT D'INFORMATIQUE

Année Académique 1975-1976

Contribution à l'étude de l'implémentation d'un langage de microprogrammation de haut-niveau

Jean-Pierre Dupriez

Mémoire présenté en vue de l'obtention
du grade de licencié et maître en Informatique.

Jury : Mr J. BRUNIN
Mr M. DIEUDONNE

Au seuil de ce mémoire, nous tenons à remercier toutes les personnes qui nous ont permis de mener ce travail à bonne fin et plus particulièrement :

Monsieur BRUNIN, directeur du mémoire, qui par ses conseils judicieux a su donner une orientation enrichissante à ce travail ;

Monsieur DEMARTEAU, de l'Institut d'Informatique, qui a orienté ce travail par son expérience pratique dans le domaine de la microprogrammation ;

Monsieur DIEUDONNE, de la M.B.L.E., pour l'accueil chaleureux qu'il nous a réservé durant la phase d'élaboration et les renseignements qu'il a pu nous procurer ;

Monsieur WILLEMS, Professeur à la K.U.L., qui nous a permis d'acquérir une connaissance pratique de la VARIAN ainsi que Messieurs les Professeurs de l'Institut d'Informatique de Namur qui nous ont donné une formation de base permettant d'aborder ce travail dans d'excellentes conditions.

TABLE DES MATIERES

Pages

PREMIERE PARTIE : Recherche d'un langage de microprogrammation

Chapitre 1	Introduction et Historique	4
Chapitre 2	Problèmes liés à la création d'un langage de haut niveau	9
Chapitre 3	Orientations	17

DEUXIEME PARTIE : Fonctions du compilateur

Chapitre 1	Description générale	25
Chapitre 2	Traduction du langage source	31
Chapitre 3	Composition des micro-instructions	40
Chapitre 4	Affectation d'une adresse aux micro-instructions	58

TROISIEME PARTIE : Champ d'application et conclusions

Chapitre 1	La compilation dans le cadre d'un compilateur de langage de haut niveau	72
Chapitre 2	Conclusion et réalisations	81

BIBLIOGRAPHIE 83

ANNEXE 1 : Approche de la VARIAN

1. Généralités
2. Interdépendance des champs
3. Description de l'adressage et expression des contraintes
4. Résumé des contraintes à respecter.

P R E M I E R E P A R T I E

Recherche d'un langage de microprogrammation

Chapitre 1 Introduction et historique

1.1. Concept de la microprogrammation

1.1.1. Qu'est-ce que microprogrammer ?
1.1.2. Quelques dates

1.2. Champs d'application

1.2.1. Efficience de contrôle
1.2.2. Instructions à la demande
1.2.3. Emulation
1.2.4. Intégration hardware-software
1.2.5. Applications temps réel de contrôle de processus

1.3. Types de micro-instructions et implications

Chapitre 2 Problèmes liés à la création d'un langage de haut niveau

2.1. Etat de la question

2.1.1. Niveaux des langages de microprogrammation
2.1.2. Implémentation des langages de haut niveau
2.1.3. Exemples de langages de microprogrammation

2.2. Objectifs à atteindre par un langage de haut niveau

2.2.1. Objectifs du langage
2.2.2. Approche du modèle idéal

Chapitre 3 Orientations

3.1. Situation et objectifs du mémoire

3.2. Travail effectué

3.3. Présentation et critique du langage

3.3.1. Avertissements
3.3.2. Le langage
3.3.2.1. Opérations mémoire
3.3.2.2. Opérations arithmétiques et logiques
3.3.2.3. Opérations conditionnelles
3.3.2.4. Opérations de contrôle de séquence
3.3.2.5. Opérations de décalage

Chapitre 1

Introduction et historique

1.1. Concept de la microprogrammation

1.1.1. Qu'est-ce que microprogrammer ?

Formellement, un ordinateur est composé de cinq unités de base :
- l'unité d'entrée
- l'unité de contrôle
- l'unité de mémoire
- l'unité arithmétique et logique
- l'unité de sortie.

Le cheminement des données et des instructions machine à travers ces unités est généralement bien compris (Fig. I.1). Par contre les signaux de contrôle le sont moins sauf par le "constructeur". Ces signaux, générés dans l'unité de contrôle, déterminent le chemin des informations ainsi que "les tops d'horloge" du système.

La microprogrammation est associée à l'approche ordonnée et systématique des fonctions de l'unité de contrôle. Celles-ci comprennent :

1. la prise en charge de l'instruction à exécuter ;
2. le décodage de l'instruction machine et le contrôle de chaque "micropas" ;
3. le contrôle du chemin des données pour exécuter la dite opération ;
4. le changement d'état de la machine pour permettre la prise en charge de l'instruction suivante.

Les unités de contrôle conventionnelles réalisent ces fonctions par un assemblage relativement complexe de "flips-flops" (ex : registres,compteurs, etc ...). Au contraire, l'unité de contrôle d'un ordinateur microprogrammable comporte deux parties : la mémoire de contrôle et le décodage du contrôle. Les éléments sont bien structurés et fournissent un moyen de contrôle bien organisé et flexible (Fig. I.2). La microprogrammation est donc une technique pour implémenter les fonctions de contrôle qui sont organisées sur la base de mots stockés dans une unité mémoire. Remarquons que si cette mémoire est altérable, la microgrammation permet la modification de l'architecture du système. Le même hardware, peut apparaître comme une variété de systèmes différents et, par voie de conséquence, atteindre des possibilités optimales pour chaque application à effectuer.

1.1.2. Quelques dates

1951 L'objectif de Wilkes était de "fournir une approche d'élaboration systématique et ordonnée pour la partie contrôle de tout système de calcul" (WILKES, 1951). Il compare l'exécution des pas individuels d'une instruction machine avec l'exécution des instructions individuelles d'un programme, d'où le terme microprogrammation.

1956/7 Glantz et Mercer R.J. (GLANTZ, 1956 ; MERCER, 1957) mettent en évidence que, par la microprogrammation, l'ensemble des instructions peut être variable.

1964 Des articles sur la microprogrammation apparaissent. Ils montrent comment on peut étendre les possibilités des

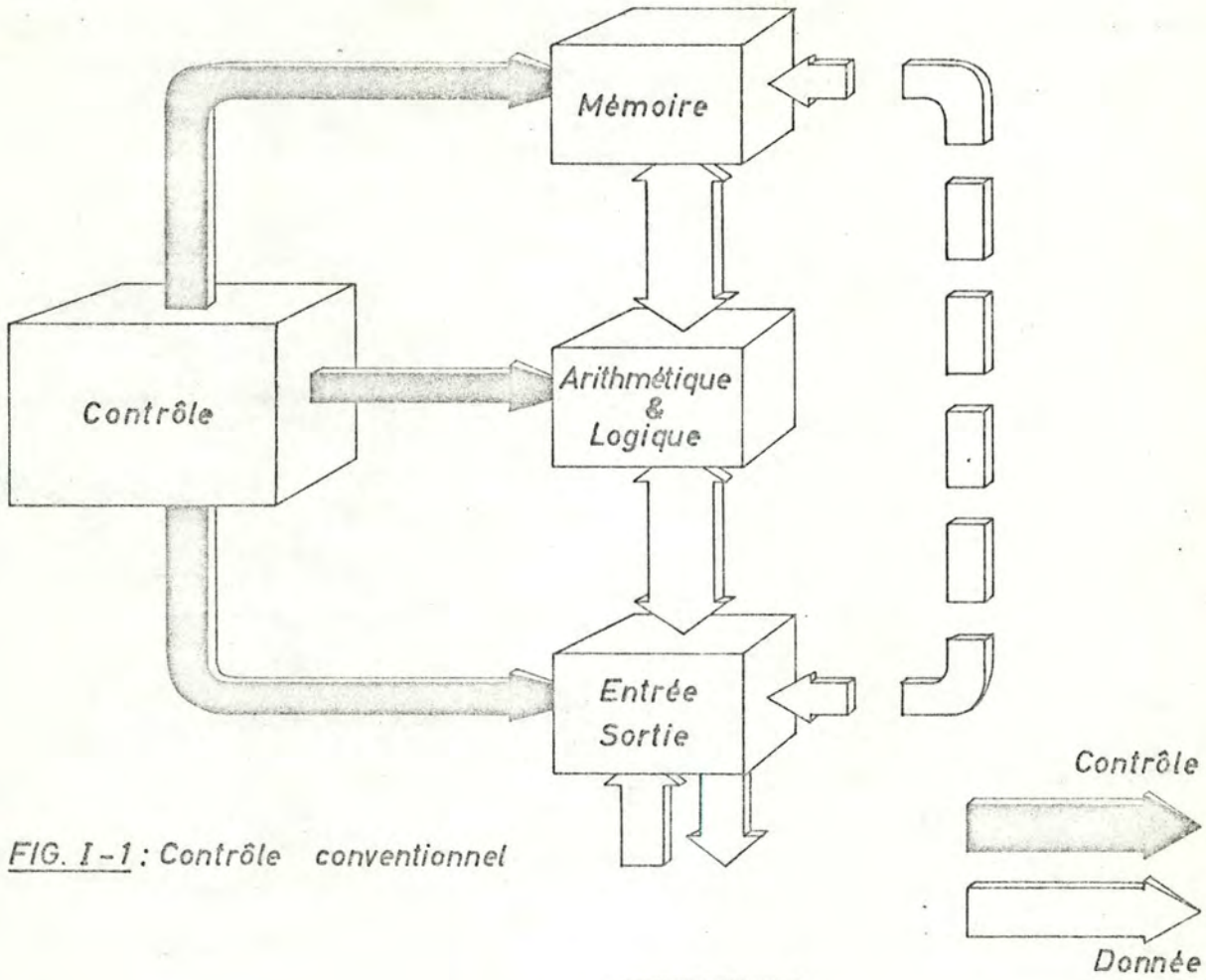


FIG. I-1 : Contrôle conventionnel

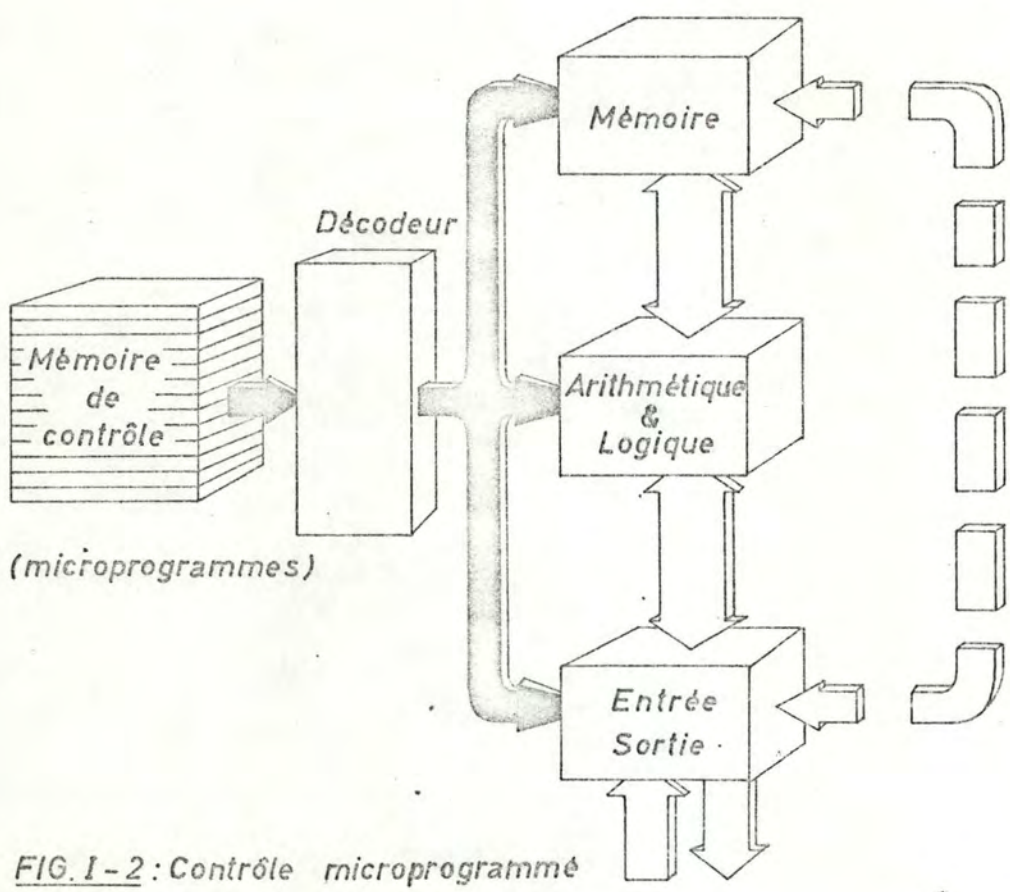


FIG. I-2 : Contrôle microprogrammé

petites machines (BECK, 1964 ; BOUTWELL, 1964 ; AMDAHL, 1964 ; HILL, 1964 ; Mc GEE, 1964). Les machines IBM 360 montrent que par la microprogrammation, des ordinateurs de puissance différentes peuvent être fournis avec un ensemble d'instructions identique.

1965 Melbourne et Pugmire (MELBOURNE, 1965) décrivent un support de microprogrammation pour "compiler" et interpréter les langages de haut-niveau.

1966 Green et Tucker (GREEN 1966 ; TUCKER, 1965) décrivent l'émulation d'une machine sur une autre par la microprogrammation.

1967 Opler (OPLER, 1967) forme le terme "firmware" pour désigner des microprogrammes destinés à supporter du software.

1969 Wilkes (WILKES, 1969) et Rosin (ROSIN, 1969) font le point sur le développement de la microprogrammation.

1970 Husson (HUSSON, 1970) publie le premier traité sur la microprogrammation.

Durant les cinq dernières années peu d'articles de base furent publiés. En fait, on s'est penché sur le développement matériel des multiples idées émises et les articles sont plutôt devenus des descriptions de réalisations pratiques.

1.2. Champs d'application

Nous pouvons distinguer cinq grandes classes d'applications. Ces classes ne s'excluent pas mutuellement ; au contraire leur intersection est significative. Ce sont les problèmes spécifiques qui se présentent qui ont amené une telle classification.

1.2.1. Contrôle efficient pour une architecture bien spécifique

Un exemple est "l'implémentation" de la gamme IBM 360 dans laquelle de nombreux processus d'organisation interne radicalement différente coexistent grâce à une microprogrammation assurant une spécification architecturale identique. C'est la compatibilité de bas et de haut de gamme. La microprogrammation est ici considérée comme un moyen économique de standardisation.

1.2.2. Adaptation d'une architecture existante avec ou sans modification du chemin des données

Cette application est plus tournée vers l'utilisateur que toutes les autres. Les concepts d'ensembles d'instructions particulières et d'addition de "macros" pour étendre l'architecture d'un système, peuvent sans problème entrer dans cette classe.

Nous pouvons qualifier cette classe d'applications de microprogrammation interprétative. En effet, le chemin de données existant est adapté pour interpréter et exécuter une autre architecture ou un langage de haut niveau. Les émulateurs, pourrait-on nous objecter, sont aussi interprétateurs, c'est vrai. Cependant, deux raisons nous guident à les placer dans une autre classe d'applications. D'abord, la conception d'émulateurs s'est considérablement étendue, ensuite les émulateurs ont de nombreux problèmes particuliers qui doivent être discutés séparément.

1.2.3. Emulation d'hardware existant

Comme nous l'avons écrit ci-dessus, nous pouvons considérer cette application comme une sous-classe de la précédente. De nombreux exemples existent ; citons notamment la VARIAN 620 émulée sur la VARIAN 70.

1.2.4. Intégration du hardware et de différents niveaux de software en un système global et intégré

C'est surtout parmi les macros IOCS, les contrôleurs et les traitements d'interrupt que l'intégration peut s'effectuer. Jacquemart étudie dans son mémoire de 1975 les problèmes relatifs à ce type d'application (JACQUEMART, 1975).

1.2.5. Application temps réel pour le contrôle de processus

Dans ce cas, le problème principal est de permettre à l'ordinateur de contrôle de processus de répondre, en temps réel, à des fonctions interdépendantes ou indépendantes. Le processus doit être suffisamment puissant pour pouvoir répondre à toutes les demandes même aux heures de pointe. Il peut être intéressant d'introduire des programmes indépendants du temps afin d'utiliser les temps morts. Le programme de contrôle doit donc être capable d'interrompre à tout instant de tels programmes afin de prendre en charge les demandes temps réel. Ce sont ces objectifs qui ont motivé la conception de la gamme VARIAN.

1.3. Types de micro-instructions et implications

Les ordinateurs microprogrammables peuvent être groupés en deux classes d'après le format des micro-instructions qui les contrôlent, celles-ci pouvant être verticales ou horizontales. A travers la littérature, ces deux termes prennent toutefois des sens différents.

Nous dirons que les micro-instructions verticales effectuent des opérations simples comme le chargement, l'addition, le sauvetage ou le branchement. Elles ressemblent souvent aux instructions du langage machine contenant un code opération et des opérandes. La longueur des micro-instructions verticales varie généralement de 12 à 24 bits.

Les micro-instructions horizontales, au contraire, contrôlent de nombreuses ressources qui travaillent en parallèle. Si les micro-instructions horizontales ont l'avantage d'utiliser le matériel de façon efficiente, la construction de microprogrammes utilisant les ressources de façon optimale est un problème difficile. Puisque la micro-instruction horizontale contrôle de multiples ressources, elle contient plus d'informations qu'une micro-instruction verticale et a donc une longueur supérieure ; une longueur de 64 bits ou plus est classique pour ce type de micro-instruction. La caractéristique déterminante pour différencier les micro-instructions verticales et horizontales est donc le nombre de ressources contrôlées simultanément.

Avec les micro-instructions à format vertical, la concurrence entre opérations est minimale. Un format horizontal cherche à donner à l'utilisateur plus de puissance en fournissant un micromot encodé de façon concise et une plus grande simultanéité d'opérations. L'optimalisation du microprogramme de format vertical est par conséquent similaire à celle d'un langage de base (type assembler). Les micro-instructions sont exécutées séquentiellement vu leur encodage maximal. Pour les machines à format horizontal,

la réduction du nombre de micro-instructions implique un autre problème : la fusion des micro-opérations individuelles en micro-instructions.

Une technique intéressante, appelée contrôle résiduel (FLYNN et ROSIN, 1971) combine les schémas verticaux et horizontaux pour réaliser, à certains points de vue, les avantages des deux méthodes. Dans le contrôle résiduel, il existe plusieurs registres "à positions" qui contiennent des informations statiques, c'est-à-dire des informations qui restent inchangées durant l'exécution de plusieurs micro-instructions. Ces registres "à positions" contrôlent plusieurs ressources, mais sont choisis et manipulés par des micro-instructions verticales. Le contrôle résiduel travaille avec des micro-instructions verticales en conservant le contrôle de ressources en parallèles par des registres "à positions".

A l'heure actuelle, si certains constructeurs,encore trop peu nombreux, permettent à l'utilisateur d'adapter lui-même un ensemble d'instructions en fonction du temps et de ses applications (voir 1.2.2. adaptation d'architecture), les software de microprogrammation sont inexistants ou à l'état embryonaire. C'est l'étude d'une partie d'un tel software, appliqué à la microprogrammation horizontale, qui fait l'objet du mémoire.

Chapitre 2

Problèmes liés à la création d'un langage de haut niveau

2.1. Etat de la question

Dans ce paragraphe, nous passerons en revue les différents types de langage de microprogrammation de haut niveau en les classifiant.

Comme pour les caractéristiques hardware, les aspects des langages de microprogrammation sont liés à leur présentation et à leur "implémentation". La présentation d'un langage de microprogrammation détermine son niveau qui peut aller d'une représentation mnémonique des micro-instructions jusqu'aux langages indépendants du hardware.L' "implémentation" d'un langage de microprogrammation décrit comment les microprogrammes exprimés dans ce langage sont effectués.

2.1.1. Niveaux des langages de microprogrammation

Comme pour les langages de programmation, il existe une variété de langages de microprogrammation de niveau de complexité différente. Au bas de l'échelle, se situent les microlangages dans lesquels chaque micro-instruction est une série de bits. Les langages d'assemblage de microprogrammation fournissent la possibilité d'exprimer les micro-instructions symboliquement comme le font les langages d'assemblage traditionnels. Dans les langages par ordinogrammes, l'enchaînement des "cadres" représente le microprogramme. Chaque cadre décrit une micro-instruction et des flèches connectrices indiquent leur séquence. Dans le cadre, chaque ligne désigne une opération ou un champ de la micro-instruction. Les langages à transfert de registres permettent l'écriture des micro-instructions dans un format fort semblable à celui des langages de programmation de haut niveau.

Les micro-opérations ou commandes qui constituent les micro-instructions représentent des opérations primitives de la machine et apparaisent comme de simples instructions dans les langages de haut niveau. Bien que, sur le plan syntaxique, ces langages soient attirants, ils ne sont que des langages d'assemblages et se montrent aussi exigeants qu'eux : la programmation demande une connaissance familière des caractéristiques hardware ; il y a une correspondance "un-un" entre les instructions du langage de haut niveau et les instructions machine et il n'y a que peu de types de variables.

Les langages à transferts de registres sont souvent utilisés pour des machines horizontales ; par contre, les langages d'assemblage et d'ordinogrammes y sont proscrit par suite du grand nombre de champs dans les micro-instructions. Les macros langages de microprogrammation à transferts de registres peuvent fournir, en plus, des équivalences générales et les facilités des macros. Ils permettent ainsi une représentation mnémonique des registres, des micro-opérations et des micro-instructions ainsi qu'une substitution paramétrée. Dans les langages de micro-programmation orientés procédures mais dépendants de la machine, les registres et les opérations peuvent être exprimés comme dans les langages à transferts de registres. Il existe d'autres facilités de langage telles que le contrôle du chemin des données, les expressions composées et une définition de la structure des données. La conver-

sion de ces facilités en micro-instructions entraîne souvent une perte d'efficacité.

La figure I-3 résume le présent paragraphe en donnant les différents niveaux ainsi que le gain obtenu en passant de l'un à l'autre. Des exemples des différents niveaux de langages sont présentés et discutés ci-dessous (2.1.3).

2.1.2. Implémentation des langages de micro-programmation

L'implémentation des langages d'assemblage et à ordinogrammes est directe : elle est généralement "un-un", c'est-à-dire qu'à une instruction correspond une micro-instruction du micro-code. Du fait de leur similarité avec les langages d'assemblage, les langages à transferts de régistres sont aussi"implémentés"par traduction directe en microcode. La traduction de ces langages est souvent effectuée par un "cross assembler" c'est-à-dire que les microprogrammes sont traduits sur un ordinateur différent et le micro-code est ensuite chargé dans la machine exécutant le microcode. Cette traduction séparée est utilisée pour des raisons pratiques. Elle permet un développement simultané du micro-processeur et de ses microprogrammes.

La variété d'implémentation des langages de haut niveau va de la traduction directe du programme en un microprogramme exécutable jusqu'à l'interprétation directe par microprogramme. Entre ces deux extrêmes, il existe un autre mode d'implémentation : la traduction (par software ou par firmware) du langage de haut niveau en langage intermédiaire (fig. I-4) interprété par microprogrammes. Cette dernière technique sera illustrée dans la 3ème partie. Le niveau du langage intermédiaire varie de l'assembleur jusqu'au codage simple du langage de haut niveau en passant par les quadruples et la notation polonaise.

2.1.3. Exemples de langages de microprogrammation

a) Langage à notation ordinogrammes.

L'approche d'IBM est sans doute la plus connue. Le CAS (Controls Automation System) a été développé pour aider le microprogrammeur dans la préparation de la logique en fournissant notamment des ordinogrammes de ces microprogrammes.

Chaque micro-instruction est décrite dans un cadre où les différentes lignes expriment soit une constante, soit le contrôle de l'ALU ou de la mémoire, etc ... Les sous-commandes sont sous forme mnémoniques, mais très proches de la notation algébrique.

exemple : (1) J + 0 + 1 → JC

(2) I + 0 + 1 → IC

(3) IJ → MN

(4) WRITE

L'instruction (1) incrémente le registre J d'une unité, sauve le report (c) qui sera propagé dans le registre I (2). La sous-commande (3) sauve les registres I et J dans les registres adresse mémoire M et N. Finalement (4) écrit à cette adresse. La séquence est indiquée par les lignes joignant les différents cadres. Pour trouver la séquence correcte, on est aidé par des "leg identifiers".

	Types de langages	Apports nouveaux
Haut niveau	Procédural, indépendant	Généralisation : création d'une machine fictive unique
	Procédural dépendant de la machine	
	"Macro" transferts de registres	Aspect haut-niveau. Connaissance partielle du hardware
Bas niveau	Transferts de registres	Regroupement de micro-instructions sous un même mnémonique
	Assembleur et ordinogrammes	Introduction de mnémoniques pour exprimer des "micro-opérations"
	Microlangage	Introduction de mnémoniques pour exprimer des groupes de bits

Figure I-3 : Niveaux des langages de microprogrammation

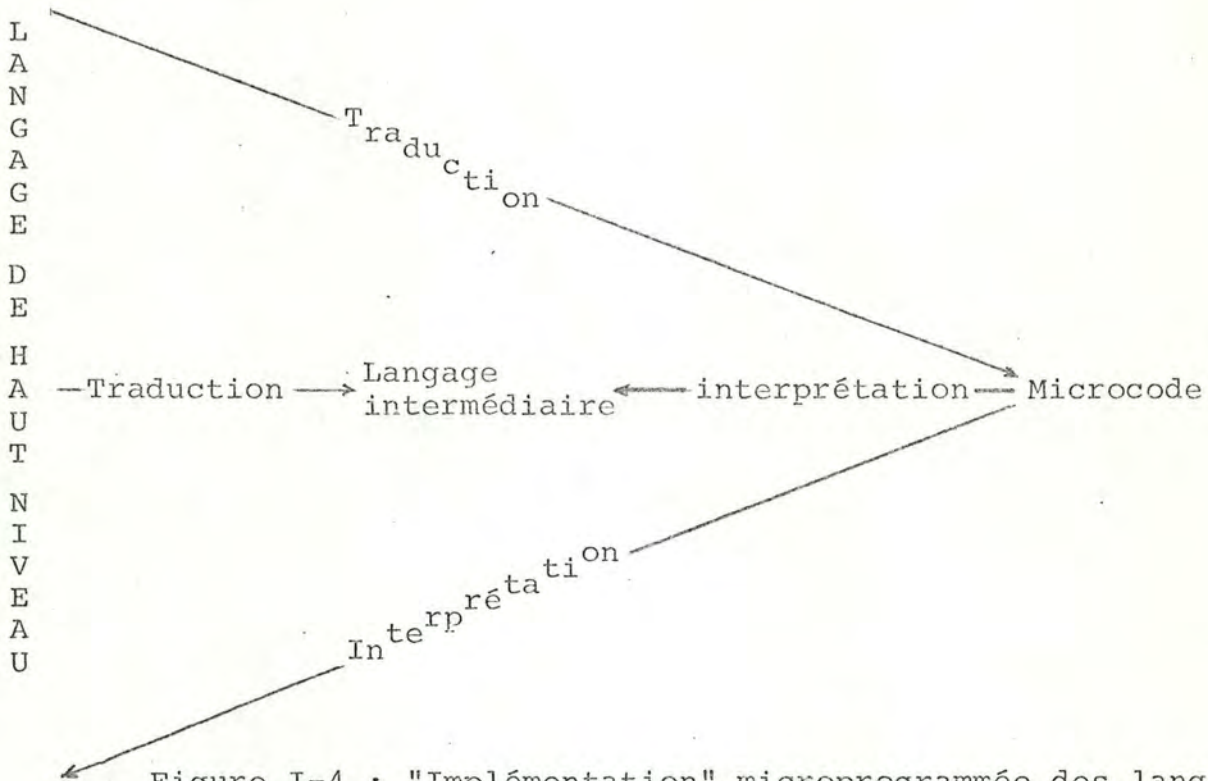


Figure I-4 : "Implémentation" microprogrammée des langages de haut niveau

Le CAS fournit ses services pour une grande variété de processeurs ainsi que pour des contrôleurs de disques. Les formats diffèrent dans leur longueur, mais non dans leur structure ce qui permet au CAS de fournir un langage unique pour tout une gamme de processeurs.

Un point de vue différent a été adopté dans le développement des microprogrammes des grands ensembles tel le Siemens 4004/150. Comme illustration, la figureI-5 montre en haut le format de la micro-instruction et en-dessous le cadre correspondant de la notation ordinogramme. Les mêmes abréviations sont utilisées dans les deux parties de la figure, car chaque champ de contrôle de la micro-instruction est repris de façon unique dans le cadre. La notation repose presque exclusivement sur des mnémoniques indiquant les valeurs non nulles des champs. Son principal avantage est la correspondance "un-un" des mnémoniques avec les champs de la micro-instruction. C'est le nombre réduit de champs (14 dont 5 pour le contrôle de séquence) qui permet une telle représentation. En reprenant la définition précédemment donnée, on peut dire que nous sommes en présence d'une microprogrammation verticale.

Ces deux approches différentes de la notation à ordinogrammes ne peuvent être appliquées que dans leur environnement particulier. Dans les autres cas, les inconvénients deviennent trop nombreux : manque de souplesse (BARTOW, 1968) spécialement du point de vue correction, documentation trop peu importante (HUSSON, 1970) et nécessité de connaître à fond le hardware pour lequel le travail est effectué.

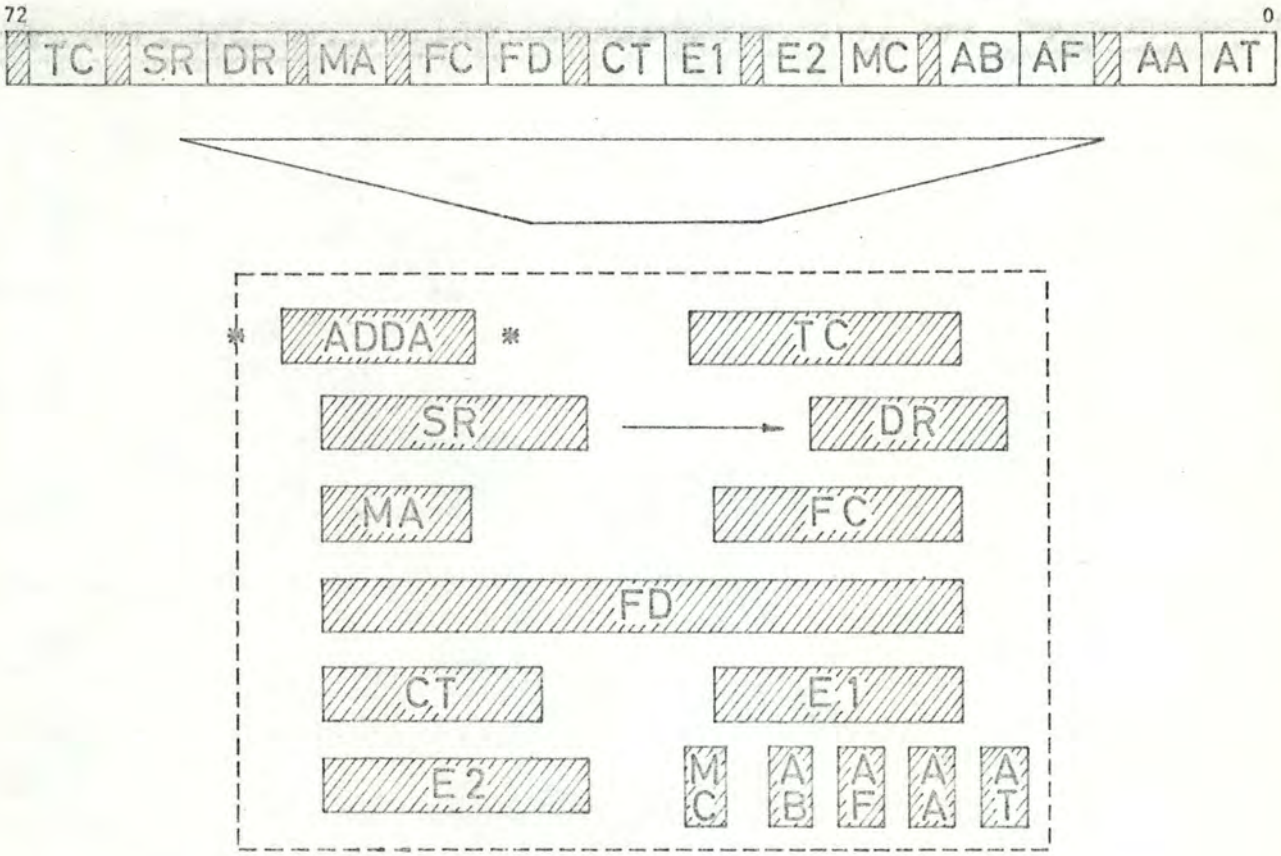
b) Langage de haut niveau dépendant du matériel

Les langages de microprogrammation pour le Data Saab FCPU, le MLP-900 et le MIL (Micro Implementation Language) développé pour le Burroughs B 1700 fournissent les instructions issues des langages de programmation par ex : IF ... THEN ... ELSE la boucle DO, l'instruction CASE, la structure de bloc BEGIN ... END, etc ... Le MPL (Microdata Programming Language) développé pour le Microdata 32/S est plus sofistiqué que les précédents. Il fournit, notamment des facilités pour allouer des places mémoires tandis que dans les autres on ne pouvait que réutiliser des registres hardware. De ce fait il n'est pas étonnant que le MPL ne soit pas traduit directement en micro-instructions.

Il est intéressant de noter que tous ces langages ont été développés pour des machines à structure verticale (pas de problème de simultanéité) en appliquant simplement des techniques déjà bien éprouvées.

c) Langages de haut niveau indépendant de tout hardware

Comme nous l'avons signalé, de tels langages appartiennent encore au domaine de la recherche. Husson (HUSSON, 1970) a présenté quelques caractéristiques que devrait avoir un langage de haut niveau. Malheureusement, à l'époque, l'expérience manquait et ses souhaits sont quelque peu utopistes. Tirell (TIRELL, 1973), trois ans plus tard, a mis en évidence les buts d'un compilateur pour la traduction d'un langage de haut niveau en



TC	Test Condition	E1	Exception Group 1
SR	Source Register	E2	Exception Group 2
DR	Destination Register	MC	Mode Control
MA	Fast Memory Address	AB	Address B
FC	Function Control	AF	Address False
FD	Function Destination	AA	Address A
CT	Counter Trigger	AT	Address True

DESIGNATION DES CHAMPS

▨ : bits de parité

ADDR : Current Address

FIG. I-5 : Format de la micro-instruction et notation ordinogramme du Siemens 4004/150

microcode et ce de façon optimisante (facilité d'écriture, de documentation, de correction et de compatibilité). L'approche de l'assignation unique (single assignent approach) a également été étudié pour certaines architectures. SIMPL (single identity microprogramming language) créé par Ramamoorthy (RAMAMOORTHY, 1974) en est un exemple.

2.2. Objectifs à atteindre par un langage de haut niveau

2.2.1. Objectifs du langage

Pour que la microprogrammation soit effectivement utile à "l'utilisateur moyen", le langage proposé doit présenter des moyens réels pour faciliter la pratique de la microprogrammation. Les langages de haut niveau sont les plus efficaces pour réduire les difficultés de codage. Un langage de haut niveau devrait :

+ Permettre à l'utilisateur d'écrire des microprogrammes d'une manière
  - conventionnelle (c'est au compilateur de reconnaître et d'exploiter les contraintes de parallélisme, donc d'optimaliser le microcode en utilisant au mieux les possibilités offertes)
  - procédurale (capable de calculer des expressions arithmétiques et logiques quelconques en les transformant en une évaluation pas à pas)
+ Permettre que ces microprogrammes soient compilés en microprogrammes exécutables et efficients.
+ Pouvoir être adapté aux besoins futurs de la machine.
+ Etre bien défini du point de vue de la sémantique.
+ Etre indépendant de la machine. Bien qu'il semble impossible de construire un compilateur capable de fournir un output accepté par tous les micro-processeurs, il faudra essayer de le faire accepter par une large gamme de machines.
+ Fournir un maximum de renseignements pour aider l'utilisateur dans son travail.

En résumé, les propriétés attendues d'un langage de haut niveau pour la microprogrammation doivent être un compromis entre :

1. La dépendance de la machine.
2. La facilité de détection et de représentation du parallélisme.
3. Le naturel demandé à tout langage de programmation pour établir des communications effectives et simples entre l'homme et la machine.

2.2.2. Approche du modèle idéal

La plupart des compilateurs décrits dans la littérature (GRIES, 1971 ; LLYOD, 1974) séparent la traduction en deux ou plusieurs phases (voir fig. I-7). Dans la phase initiale, l'analyse syntaxique et sémantique produit un langage intermédiaire (LI, d'une machine abstraite). Dans les phases suivantes le LI est transformé en des langages intermédiaires de niveaux de plus en plus bas jusqu'à la génération des instructions exécutables.

Du fait de la performance exigée des microprogrammes, nous devons générer un microprogramme optimisé. Ce qui implique qu'à certains niveaux de la compilation nous devons reconnaître les actions concurrentes ainsi que les contraintes de ressources. Après cette identification, les actions sont regroupées dans une microinstruction horizontale de façon à ne pas modifier

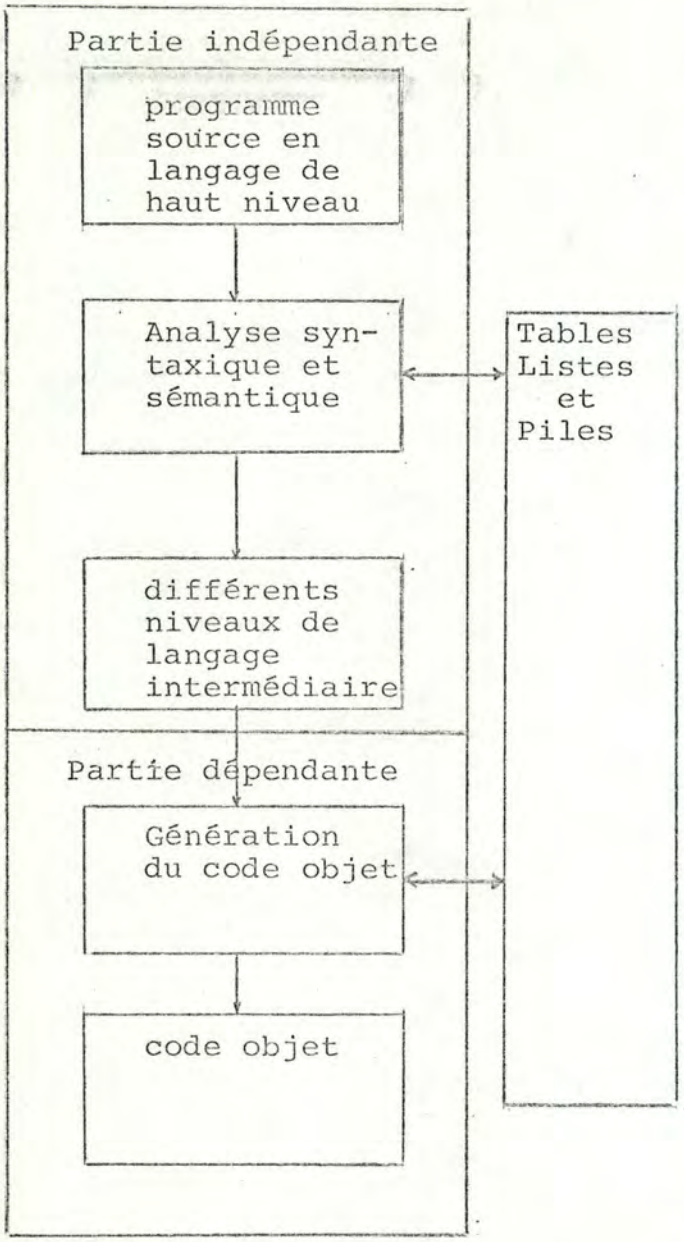


Figure I-7 : modèle élémentaire de compilateur d'un langage de haut niveau classique.

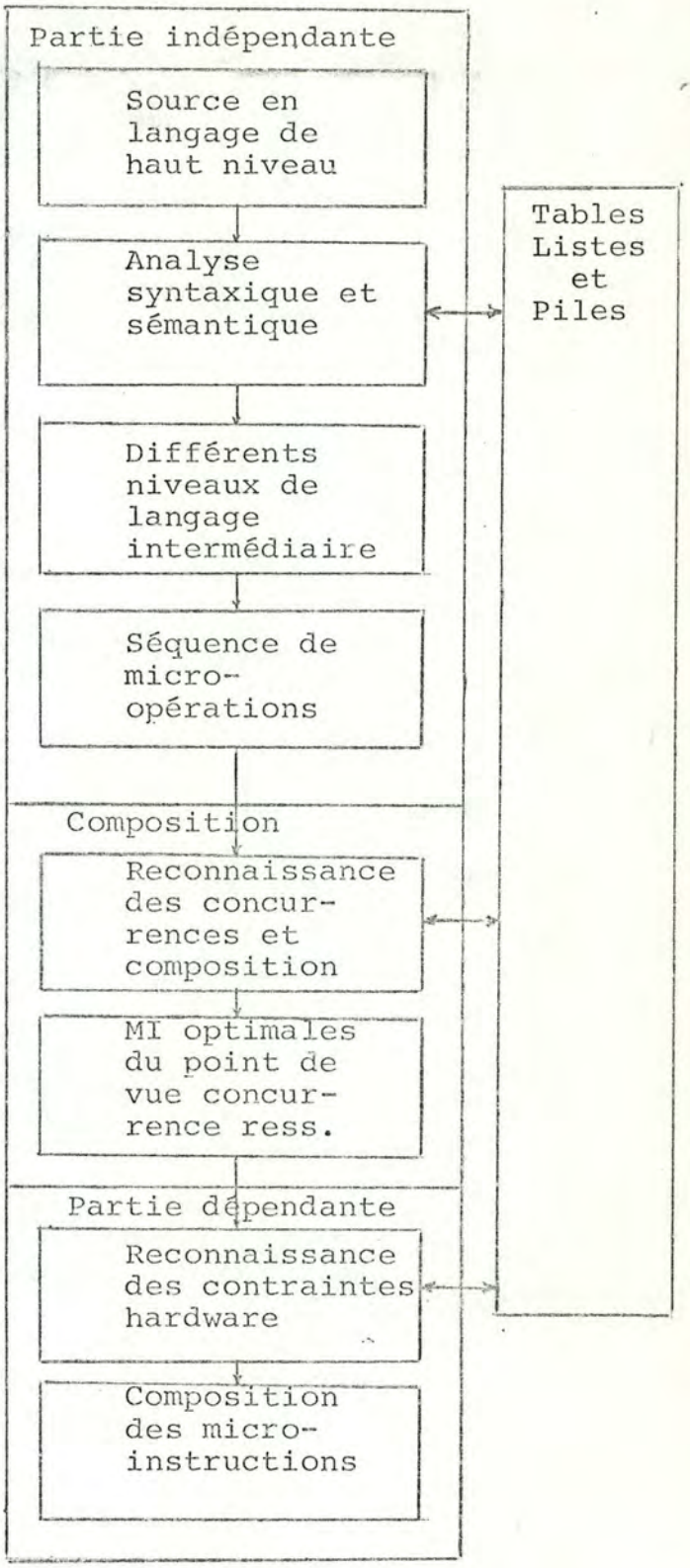


Figure I-6 : modèle de compilateur d'un langage de microprogrammation de haut niveau.

la logique du programme. Un modèle de compilateur pour langage de microprogrammation de haut niveau pourrait être décomposé en deux parties principales : la première, l'analyse syntaxique et sémantique produisant un langage intermédiaire ; la seconde transformant le langage intermédiaire en séquence de micro-opérations (MOP) pouvant être regroupées dans des micro-instructions (MI).

Le modèle proposé à la figure I-6 solutionne quelques problèmes et rencontre les objectifs décrit ci-dessus. Dans la partie indépendante nous proposons un traducteur complet qui transforme, à travers plusieurs niveaux, le langage source en un LI qui est une séquence de MOP pour un micro-processeur parallèle fictif. Ce LI devrait être suffisamment général pour rencontrer les propriétés générales des différentes machines. Pour atteindre une efficacité maximale, la syntaxe du langage de haut niveau (LHN) peut être basée sur ce LI général. Ensuite, vient une recherche des concurrences de dépendances dans l'utilisation des ressources et un algorithme d'optimisation compose une nouvelle séquence de MOP. La génération de code (partie complètement dépendante de la machine) décompose éventuellement les regroupements de MOP pour tenir compte des contraintes spécifiques à l'installation.

Chapitre III

Orientations

3.1. Situation et objectifs du mémoire

Lors de l'analyse du problème de la création d'un langage de haut niveau, nous avons établi un modèle idéal de compilateur (point I.2.2.2). La réalisation de ce modèle pose de nombreux problèmes de types différents dont certains sont classiques et se rencontrent lors de l'établissement de n'importe quel nouveau langage. Comme toujours deux méthodes de travail se présentent : l'une "top-down" qui consiste à partir de la définition d'un langage de haut niveau, de sa grammaire pour arriver à un microcode en passant par des langages intermédiaires; l'autre "bottom-up" entreprend l'étude par le biais d'une définition de machine. Elle crée d'abord un langage intermédiaire indépendant traductible en micro-instructions et aboutit à la définition d'un langage de haut-niveau. Le point commun, qui est en fait l'interface entre les deux grandes parties du compilateur, serait donc une espèce de macrolangage ou ler langage intermédiaire (= LIM dans le modèle).

La réalisation complète du compilateur comprend deux parties mobilisant chacune les ressources dont nous disposons pour l'élaboration de ce mémoire. Un choix doit donc s'effectuer. Le risque de l'approche "top-down" est la création d'un langage intermédiaire non traductible en micro-instructions. L'autre risque par contre de créer un langage intermédiaire ne pouvant être généré par un automate. Compte-tenu de ces risques, l'approche "bottom-up" a été choisie pour différentes raisons :

- un micro-processeur possédant une mémoire de contrôle modifiable était à notre disposition, l'étude serait donc concrète ;
- l'étude de problèmes nouveaux, spécifiques à la microprogrammation a été jugée préférable à une étude de problèmes classiques ;

Afin de répondre à l'objection que le langage intermédiaire est incomplet et non cohérent, un exemple d'écriture par automate a été étudié et présenté en 3ème partie.

C'est donc la création d'un langage intermédiaire, la recherche des concurrences et l'optimalisation du microcode qui seront étudiées dans la suite. En bref, nous avons voulu construire un langage intermédiaire (LIM) à la fois utilisable par le microprogrammeur et indépendant de la machine.

3.2. Travail effectué

La création et la traduction en microcode du macrolangage ou premier langage intermédiaire ont été effectuées pour la VARIAN 73, machine à microinstructions horizontales. Le microcode généré a été optimalisé du point de vue occupation mémoire en utilisant des algorithmes de détection de parallélisme. Le microcode est chargeable en mémoire, c'est-à-dire que l'adressage des micro-instructions peut être effectué.

Nous ne présenterons pas certains détails, même s'ils ont été étudiés et élucidés, comme la fusion des adresses et des micro-instructions ou la mise en forme "chargeable" du microprogramme.

Le micro-assembleur élaboré tient compte d'un certain nombre de contraintes hardware. Cependant, pour des raisons de performances, d'autres contraintes sont laissées au niveau du langage. C'est ainsi que la synchronisation avec la mémoire reste à l'utilisateur. Ces quelques exceptions seront signalées dans le cours de l'exposé afin de mieux saisir les motifs qui nous y ont amené.

3.3. Présentation et critique du langage

3.3.1. Avertissements

Dans le paragraphe suivant, nous décrivons le langage intermédiaire. Par rapport au modèle général analysé au point 2.2.2., ce langage intermédiaire est celui qui doit être généré par partie indépendante de tout hardware.

Pour la bonne compréhension de tout microlangage, il faut connaître le chemin des données et des contrôles. A cet effet l'annexe 1 rassemble les principales caractéristiques de la VARIAN ainsi qu'un schéma descriptif du chemin des données.

Nous ne décrivons aucune instruction fournissant des facilités (de corrections, de contrôles) à l'utilisateur. Si ce point est des plus importants dans l'établissement d'un assembleur, sa réalisation ne présente aucune difficulté logique. Signalons que dans le cadre du modèle général, le niveau de définition de ces instructions est du même ordre que celui du langage de haut niveau ; et qu'en outre, il faut tenir compte de leur existence dans l'élaboration du compilateur (voir I-3.1).

3.3.2. Le langage

Le langage intermédiaire travaille au niveau de micro-opérations qui seront regroupées (partie composition) en respectant les contraintes du matériel (voir annexe 1). L'ensemble des micro-opérations a été divisé en 5 classes distinctes :

- opérations avec la mémoire ;
- opérations avec l'unité arithmétique et logique (ALU) ;
- opérations de tests ;
- contrôle de séquence ;
- opérations de décalage.

3.3.2.1. OPERATIONS AVEC LA MEMOIRE

forme générale "Opération (adresse [, $^{TESTT}_{TESTF}$])"

où [$^{TESTT}_{TESTF}$] exprime que l'opération est liée à un test posé par une instruction conditionnelle (I-3.3.2.3) "TESTT", l'opération se fera quand le test est vérifié et "TESTF" dans le cas contraire.

Opérations :

IF	(instruction fetch). Lecture d'un mot (16 bits) en mémoire centrale et sauvetage dans le MIR et l'IBR
OF	(operand fetch). Lecture d'un mot en mémoire centrale et sauvetage dans le MIR
OS	(operand store). Sauvetage d'un mot en mémoire
BS	(byte store). Sauvetage d'un byte.

Dans l'OS et le BS, la donnée à sauver se trouve sur la sortie de l'ALU. L'utilisateur doit s'assurer qu'elle s'y trouvera jusqu'au WAIT ou l'opération mémoire suivante par l'utilisation d'un TRN (data,).

WAIT synchronisation avec le cycle mémoire. Son emploi est rendu obligatoire après chaque opération mémoire si les registres mobilisés doivent être utilisés ou si la donnée lue doit être traitée

Adresse : donne le lieu où se trouve l'adresse de la mémoire

ALU la sortie de l'unité arithmétique et logique

P dans le compteur d'instructions

MIR dans le registre entrée de la mémoire

Rn dans le registre général n

OVR (override) l'adresse de l'opération est identique à celle de l'opération précédente.

3.3.2.2. OPERATIONS ARITHMETIQUES ET LOGIQUES

a) Opérations unaires

forme générale "Opérations (opérande, destination)"

opérations : INC incrémenter l'opérande avant transfert dans la destination

DEC décrémenter l'opérande avant transfert

TRN transfert de l'opérande dans la destination

TC complémenter à 2 l'opérande et le transférer

NOT complément vrai de l'opérande avant transfert.

b) Opérations binaires

forme générale : "Opérations (opérande 1, opérande 2 [,report], destination) "

opérations ADD

SUB

AND

OR ou inclusif des deux opérandes

XOR ou exclusif.

opérandes

Rn registre général n (n = 0,1 ... 15)

SRn registre général n décalé d'une position à droite

SLn registre général n décalé d'une position à gauche

P (program. counter)

MIR (memory input register)

STAT mot d'état du processus. Contient notamment tous les flags de l'ALU (signe - report ...)

MSKxxxx registre instruction masqué par la constante hexadécimale xxxx

OPR (operand register)

OLSE byte gauche de l'OPR, le signe étant étendu

ORSE byte droit de l'OPR, le signe étant étendu

OLZF byte droit de l'OPR, le byte gauche étant rempli de zéros

ORZF byte droit de l'OPR dans le byte gauche, étant rempli de zéros remplissant le droit

X'xxxx constante hexadécimale

O'xxxxx constante octale

ZERO opérande tout à zéro

ONES opérande tout à 1 (FFFF)

SC (shift counter)

RSi registre général choisi par les bits i + 3 à i du registre instruction (IR). Ce type d'opérande ne pouvant toutefois être utilisé qu'une seule fois par micro-opération.

Destinations : OPR - SC - P - Rn - IR

"blanc" le résultat se trouve uniquement affiché sur la sortie de l'ALU.

Report : Ø ou "blanc" ne pas tenir compte du report

1 considérer un report "1"

SC prendre le report sauvé auparavant

CSC considérer le complément du report sauvé

Le report étant sauvé par un "SAMPLE (ALUC)" (voir en I-3.3.2.3.).

La lecture des opérations arithmétiques et logiques fait apparaître des opérandes spécifiques à la VARIAN (ex : OLSE, ORZF, RSi ...). En fait, les opérandes prévus sont trop nombreux et les automates ne généreront qu'une partie de ceux-ci.

3.3.2.3. OPERATIONS CONDITIONNELLES

forme générale : "Opération (flag [, condition])"

opérations : TEST

SET positionnement d'un flag à la valeur 1 (SET) ou 0 (RESET)

RESET ces deux instructions ne sont valables que pour OVFL

SAMPLE échantillonnage du flag. Obligatoire avant chaque test

flags : OVLF peut être positionné à 1 ou 0 inconditionnellement. Il peut ainsi servir à contrôler des boucles. Il est automatiquement remis à zéro lors du chargement du système ou lorsque l'on spécifie un TEST (TFIR) (voir ci-dessous) et que le bit 0 de l'IR est à 1.

SSW3, SSW2, SSW1 flags positionnés uniquement manuellement par manipulation du panneau de contrôle

ALUO l'ALU est-il tout à un ?

ALUS signe de l'ALU

ALUC report de l'ALU

ALUZ l'ALU est-il tout à zéro ?

SHFT copie du bit 15 du registre général spécifié et appartenant à la liste suivante (P, Rn, le registre décalé par SRn et SLn, ZERO,ONES) si l'on n'utilise pas un littéral (X'xxxx). Il peut être testé par une micro-instruction pour provoquer un branchement suivant sa valeur (voir I.-3.3.2.4)

MIRS copie du bit 15 (signe) MIR

SFTC signale si le SC est tout à 1 (X'FFFF)

GPRS copie du bit 15 du registre 0

NORM vaut 1 après toute micro-opération qui positionne les bits 15 et 14 à des valeurs différentes. Il est remis à zéro chaque fois que les bits 15 et 14 de la sortie de l'ALU sont égaux

QUOS flag du quotient : copie du bit 15 de l'ALU après une micro-instruction n'utilisant pas de constante

MULS signe de la multiplication : positionné par toute micro-opération arithmétique dans laquelle une des conditions suivantes est vérifiée

a) le bit 15 de l'ALU et le bits 15 d'un des

opérandes sont tous deux à un
b) le bit 15 des deux opérandes est à 1

BYTA copie le bit 0 d'un registre général décalé et immédiatement utilisé dans une autre opération arithmétique. Ce flag peut être utilisé pour déterminer l'adresse de l'instruction suivante à exécuter (voir I-3.3.2.4) et dans les BS (opération mémoire) pour déterminer quel byte choisir dans le sauvetage.
Si BYTA = 0, il s'agit du byte gauche

conditions : TESTT, TESTF } utilisé uniquement pour le "TEST" et déterminent si l'on teste la condition vraie (TESTT) ou fausse (TESTF)

3.3.2.4. OPERATIONS DE CONTROLE DE SEQUENCE

JUMP (symbole) c'est la rupture de séquence inconditionnelle

JUMP (page,symbole) saut inconditionnel dans une autre page de la w.c.s. (writable control store)

GO TO (ad. vraie, ad. fausse) saut conditionnel à une condition testée par une instruction de test (I-3.3.2.3)

NOP (no operation) ne génère rien

GO TO (IR_y, MSKxxxxx, ni_1, ni_2 ... [,n_{ik}] ... n_{in})
sauf suivant la valeur des bits y à y+4 de l'IR. Ceux-ci pouvant être masqués MSKxxxxx où chaque x à la valeur 1 lorsque le bit correspondant est sélectionné. Les ni_j sont les adresses de branchement où j est égal à la valeur des bits sélectionnés quand le saut est désiré. "j" doit varier de 0 à la valeur maximale qu'il est possible de générer avec le nombre de bits sélectionnés même si les adresses ne sont pas utilisées
Ex : GO TO (IR0, MSK 10110, N1, N2, N3, N4, N5, N6, N7, N8)
où IR0 = sélection des bits 1,2,4 de l'IR
d'où 8 possibilités de branchements

DECODE branchement par l'intermédiaire du décodeur (voir technique dans l'annexe 1. Description de l'adressage). Il utilise les bits de registre instruction (IR).

CALL (page, symbole, ad-retour) saut à une adresse avec stockage de l'adresse de retour sur une pile (maximum 15 entrées)

RETURN branchement à l'adresse se situant au-dessus de la pile

GO TO (STEP, ad-1,ad-2)
GO TO (BYTA,ad-1, ad-2)
GO TO (INT, ad-1, ad-2)
GO TO (SHFT, ad-1, ad-2)
} branchements spéciaux utilisant des flags privilégiés : (voir page suivante)

STEP flag indiquant que le processeur travaille en mode pas à pas. Uniquement modifiable au panneau de contrôle

BYTA et SHFT (voir ci-dessus I-3.3.2.3)

INT flag d'interrupt que le programmeur peut remettre à zéro (RESET (INT))

3.3.2.5. OPERATIONS DE DECALAGE

forme générale : "opération (opérande, type)"

Opérations SHFTL décalage gauche d'une position

SHFTR décalage droit d'une position

Opérandes OPR le registre opérande

Rn le registre général n

Types LOGIC les 15 bits du registre spécifié sont décalés d'une position, un zéro occupant la position rendue vacante

ARITH le bit 15 ne participe pas à l'opération, un zéro occupant la position rendue vacante

DOUBLE ce type travaille avec l'OPR
- si l'opérande est un registre général, le bit "expulsé" se place dans le bit o (SHFTR) ou le bit 15 (SHFTL) de l'OPR
- si l'opérande est l'OPR, nous devons spécifier un registre général entre parenthèse. Celui-ci jouera le même rôle que l'OPR du cas précédent (ex : SHFTR (OPR, DOUBLE (R3))

ROTATE le bit 15 de l'opérande est sauvé dans le bit 0

Remarquons que la VARIAN n'effectue pas physiquement les décalages opérés sur les registres généraux (voir schéma de la VARIAN dans l'annexe 1). Le registre général est décalé et rentre dans l'ALU comme opérande. Pour avoir le décalage physique, il faut recopier la sortie de l'ALU dans ledit registre. Dans un langage intermédiaire général, il faudrait que cette opération se fasse automatiquement. Nous l'avons laissé "aux bons soins" du microprogrammeur ce qui lui permettra d'utiliser au mieux sa machine.

D E U X I E M E P A R T I E

Fonctions du compilateur

Chapitre 1 Description générale

1.1. Les différentes parties du compilateur

1.2. Données internes et externes

Chapitre 2 Traduction du langage source

2.1. Définition du langage intermédiaire

2.2. Type de traducteur

2.3. Fonctionnement du traducteur

2.3.1. Module STSTO
2.3.2. Routines de traitement
2.3.3. Exemple de traduction

Chapitre 3 Composition des micro-instructions

3.1. Buts de la composition

3.2. Techniques d'optimalisation

3.3. Options choisies

3.4. Reconnaissance des concurrences et composition

3.5. Exemple

3.6. Algorithme d'optimalisation

Chapitre 4 Affection d'une adresse aux micro-instructions

4.1. Définition du problème

4.2. Algorithme

4.2.1. Assignation des adresses
4.2.1.1. Philosophie de la procédure de résolution
4.2.1.2. Heuristique de résolution
4.2.1.2.1. Affectation d'une valeur aux bits de poids faibles
4.2.1.2.2. Différentiation des groupes
4.2.1.2.3. Affectation des micro-instructions indépendantes
4.2.2. Insertion des adresses dans les micro-instructions

Chapitre 1

Description générale du compilateur

1.1. Définition générale des différentes parties du compilateur

Reprenons notre modèle général de la figure I-6. Dans un premier temps nous avons choisi de partir du langage intermédiaire (voir I-3.1) pour arriver au microcode. Ensuite, ce langage intermédiaire a pris une forme mnémonique compréhensible par un microprogrammeur (voir I-3.3.1.). Le langage est, de ce fait, devenu micro-assembleur ; le travail suivant consiste dans l'établissement d'un compilateur dont le schéma général (fig. II-1) sera un sous-schéma de la figure I-6 auquel s'ajoute un module d'analyse syntaxique.

Le compilateur est logiquement découpé en deux parties. La première comporte une analyse syntaxique qui part du programme source et produit un langage intermédiaire. Quant à la seconde partie, elle est orientée vers les deux grands problèmes que sont d'une part, la prise en charge des contraintes hardware pour composer les micro-instructions et d'autre part, l'assignation d'adresses à ces dernières.

La figure II-2 décrit l'analyse syntaxique. Le programme source y est lu et c'est dans cette partie aussi que toutes les informations utiles par la suite (tables de symboles, tables des erreurs, ...) doivent être rassemblées. Pour limiter le problème, nous considérons que, dans cette phase, nous traitons un programme source contenant uniquement des instructions décrites précédemment. C'est donc dans une phase précédente que le problème des macros ou encore, celui des ordres à l'assembleur (du type PRINT, EJECT, ...) auront été résolus. Seules les "équivalences" et les "externes" seront pris en considération, car ils interviennent au niveau de la table des symboles. La partie analyse doit aussi :

+ Transformer les micro-opérations en un langage intermédiaire (ILTSTO) et mettre à jour la table des correspondances (CTSTO) pour permettre dans les phases suivantes d'appareiller le texte source et le langage intermédiaire;

  pour cela il faut évaluer :

  - les mnémoniques exprimant le code opération (MOTGET);
  - les opérandes (TOR_i) ;

+ Exécuter les "équivalences" et les "externes" pour mettre à jour la table des symboles (POTGET);
+ Tenir à jour la table des erreurs (ETSTO).

La seconde partie (fig.II-3) s'organise autour de la notion de blocs d'opérations (= ensembles d'opérations ayant une seule entrée et une seule sortie). La lecture de la table des symboles (STGET) permet de reconnaître facilement les blocs (BTSTO). Pour chaque bloc lu (BTGET), il faut trouver et tenir compte des concurrences de ressources (DGSTO), prendre en charge les contraintes hardware que le microprogrammeur ignore pour finalement regrouper les micro-opérations en micro-instructions (ROTGET1). Il reste alors à assigner une adresse à chaque micro-instruction (ATSTO et ROTGET2) afin qu'elles puissent toutes se trouver dans le mémoire de contrôle.

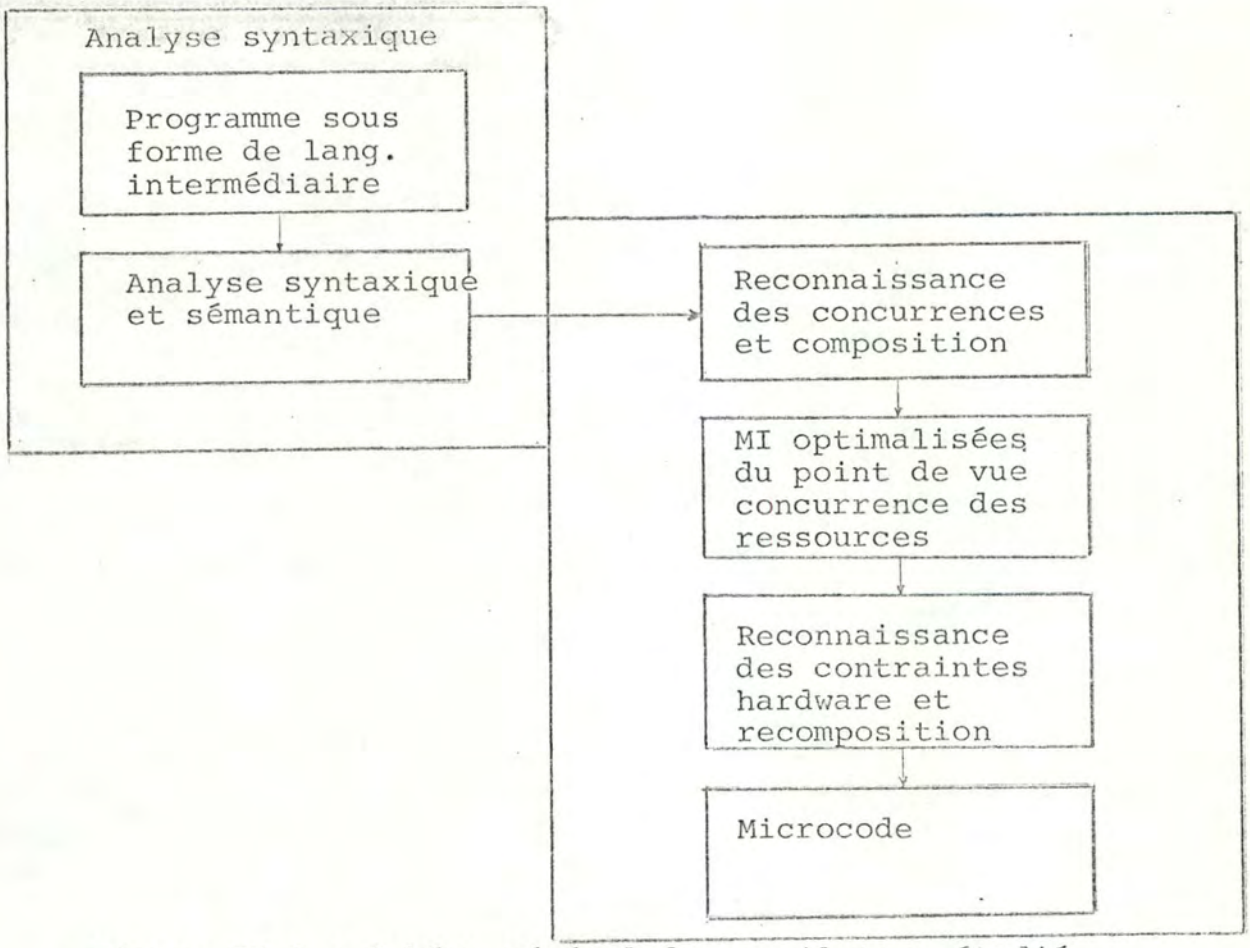
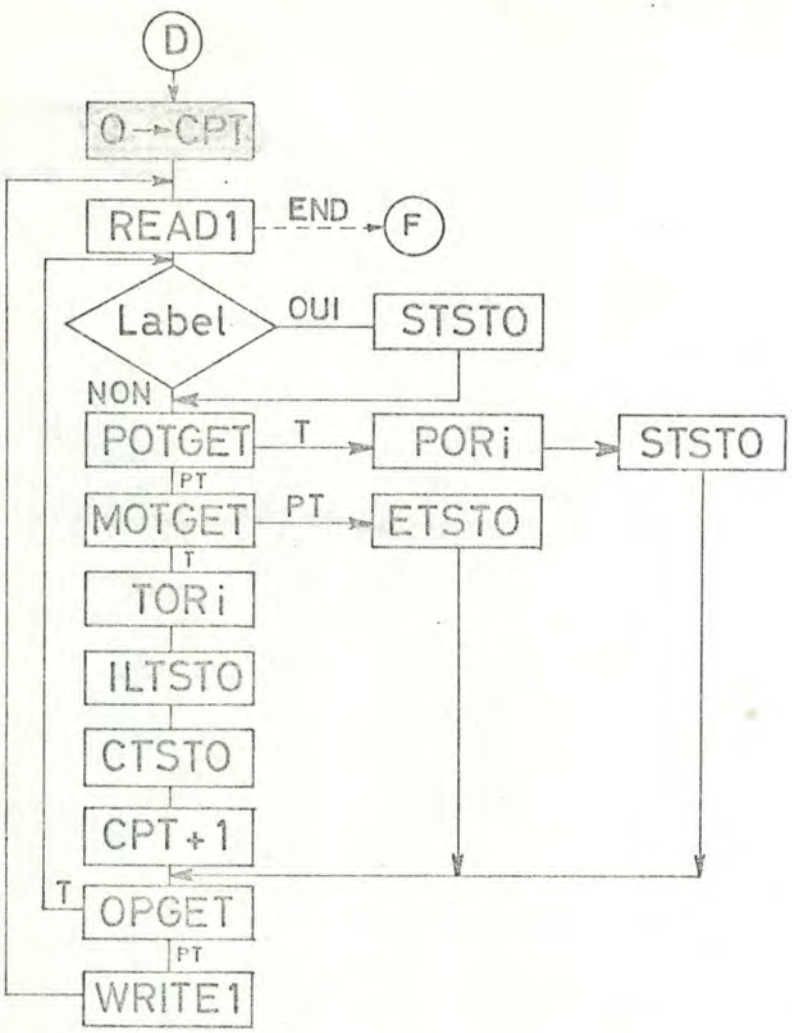


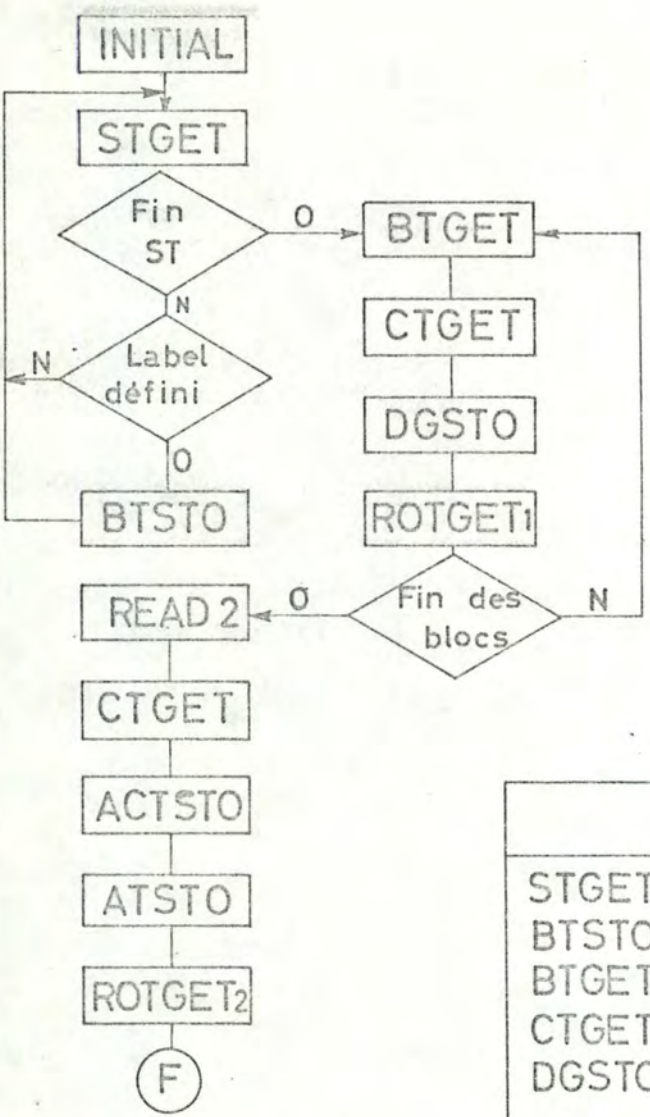
Figure II-1 : Schéma général du compilateur étudié.



CT	Compteur d'opérations
T	Opér. trouveé dans table
PT	Opér. non trouvée
TORi	Routine de traitement d'une opération
PORi	Routine de traitement d'une pseudo-opération

FONCTIONS DES MODULES	
READ 1	Lecture d'une instruction source (carte)
POTGET	Recherche du code opération dans la table des pseudo-opérations (POT)
MOTGET	Recherche du code opération dans la table des opérations (MOT)
STSTO	Mise à jour .de la table des symboles (ST)
ETSTO	Mise à jour de la table des erreurs (ET)
ILTSTO	Génération du langage intermédiaire
CTSTO	Mise à jour de la table des correspondances (CT)
OPGET	Recherche d'autres micro-opérations sur la carte
WRITE 1	Copie du langage source sur fichier SP

FIG. II-2 : Première partie du compilateur :
Analyseur syntaxique et sémantique



FONCTIONS DES MODULES	
STGET	Lecture de la table des symboles
BTSTO	Génération de la table des blocs
BTGET	Lecture de la table des blocs
CTGET	Lecture table des correspondances
DGSTO	Génération du graphe de dépendance
ROTGET 1	Optimalisation ou composition
READ 2	Lecture langage source sur fichier SP
ACTSTO	Génération table des séquences
ATSTO	Assignation des adresses
ROTGET 2	Fusion des micro-opérations et des adresses

FIG. II-3 : Deuxième partie du compilateur : Composition et adressage

1.2. Données internes et externes

Nous décrivons brièvement l'ensemble des données nécessaires au fonctionnelles des différents modules décrits ci-dessus. L'analyse syntaxique utilise :

1. un programme source (SP) ;
2. une table des codes opérations existants (MOT) indiquant le mnémonique et l'action correspondante à effectuer (TORi) ;
3. une table des pseudos-opérations (POT). Sa structure est identique à celle de MOT

et doit créer :

4. une table de symboles (ST) servant à mémoriser les symboles, le lieu de leur définition et de leur utilisation ;
5. une table des erreurs (ET) donnant le numéro de l'opération erronée et le type de l'erreur ;
6. une table de correspondances (CT) donnant le numéro de la micro-opération du langage source et le numéro de la micro-opération correspondante dans le langage intermédiaire ;
7. une copie du langage source pour impression et l'insertion des erreurs ;
8. les instructions en langage intermédiaire (ILT)

La seconde partie travaille à partir :

1. de la table des symboles (ST);
2. de la table des correspondances (CT);
3. du langage intermédiaire (ILT);

elle complète :

4. la table des erreurs (ET);

et crée :

5. une table d'adresse de micro-instructions (AT);
6. un graphe de séquences (SG) ;
7. des micro-instructions chargeables (MIT).

La figure II-4 illustre les relations entre les différentes données.

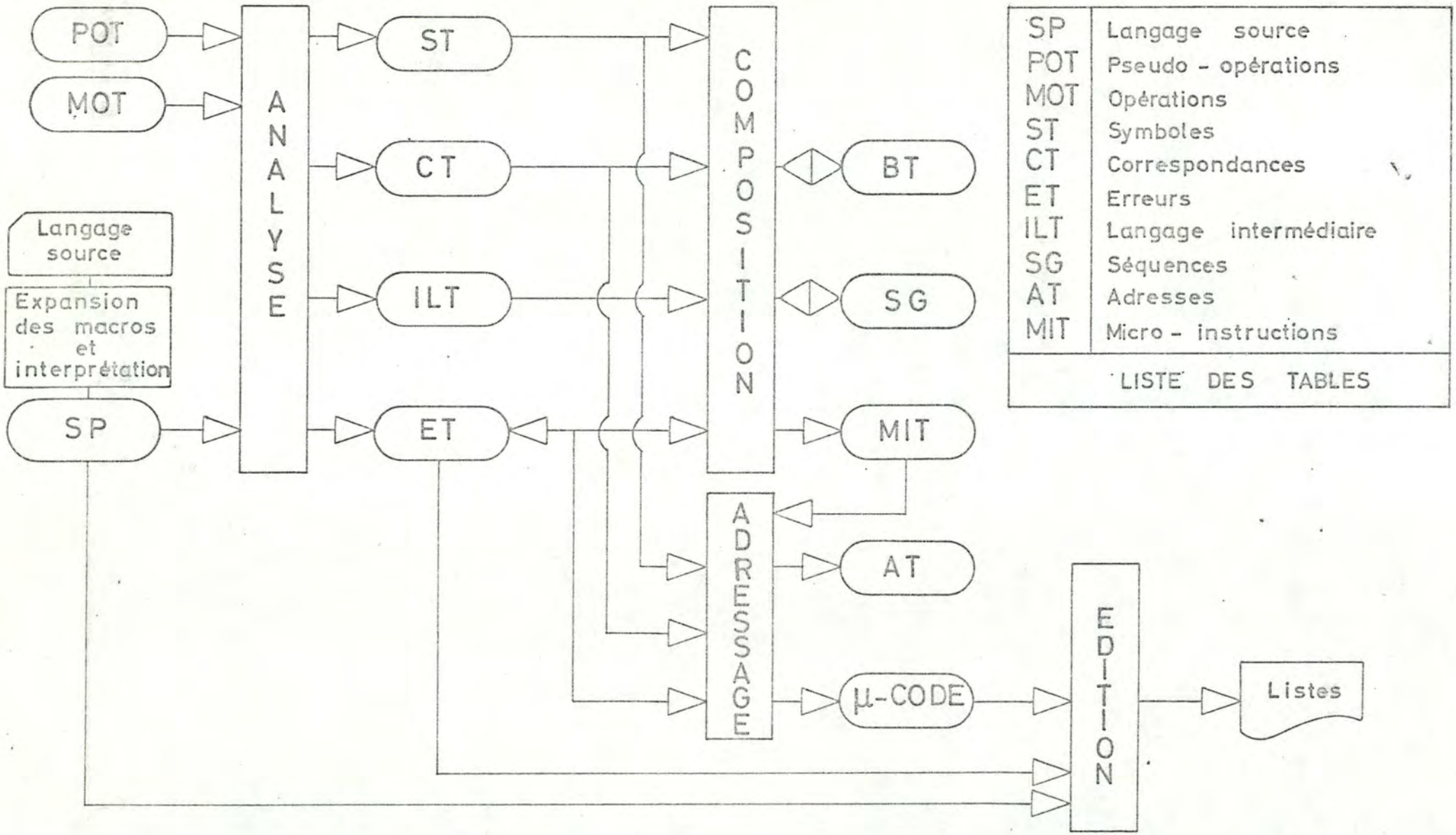


FIG. II - 4 : Relations entre les tables du compilateur

Chapitre 2

Traduction du langage source

2.1. Description et justification du langage intermédiaire (fig. II-5)

Lors de la description du langage mnémonique, nous avons distingué cinq classes d'instructions (de test, avec la mémoire, arithmétiques et logiques, de contrôle de séquence, de décalage). La micro-opération du langage intermédiaire conserve cette distinction, car d'une part le traitement de la dépendance est fonction de la catégorie et d'autre part deux micro-opérations de même type n'appartiendront jamais à la même micro-instruction. En fait, cela n'est vrai que si la notion de type est plus fine que celle de classe. Le tableau ci-dessous illustre la différence entre ces deux notions.

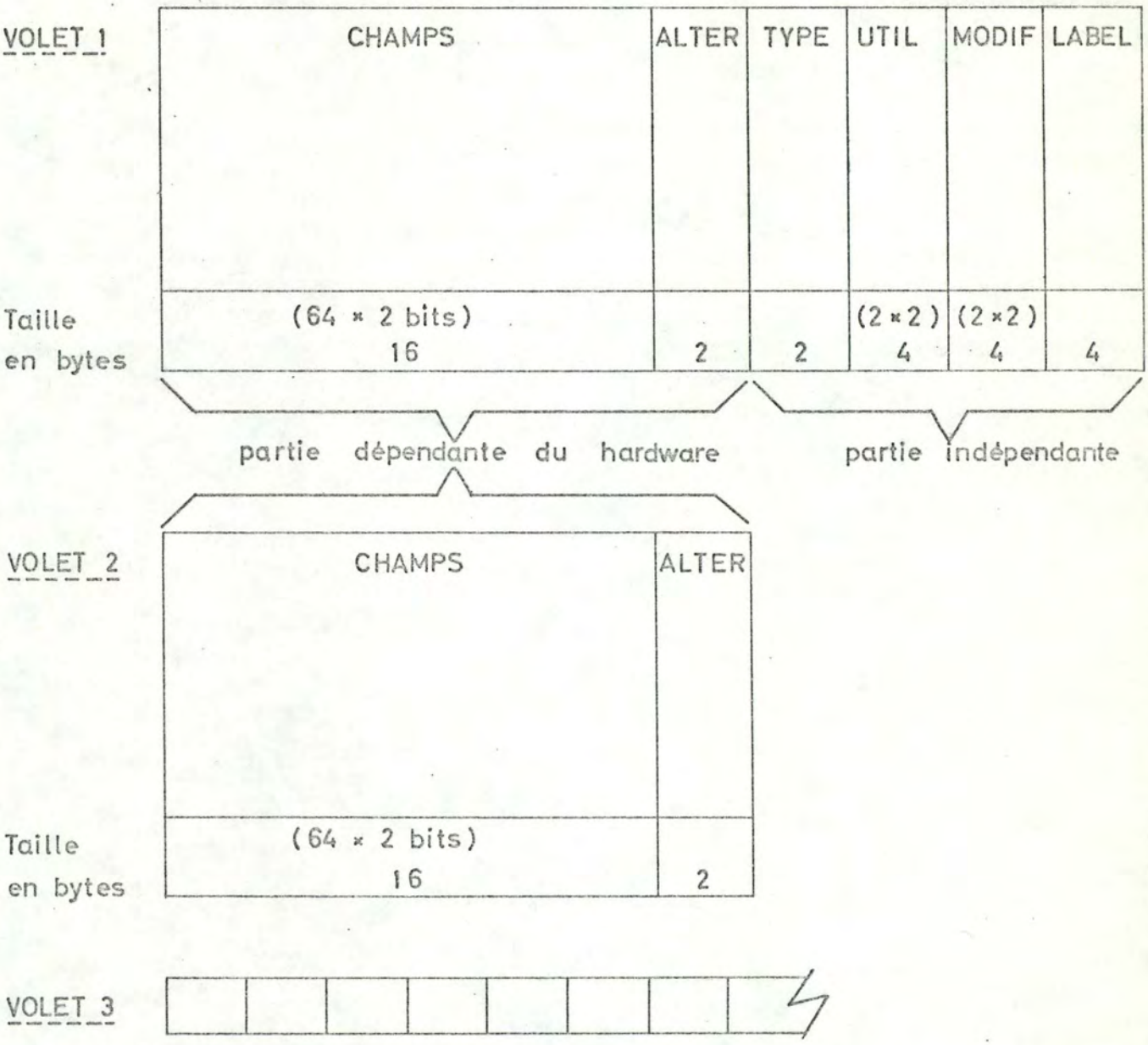
classes	code	types
TEST	1	échantillonnage (SAMPLE)
	2	test (TEST)
	3	autres (SET, RESET)
MEMOIRE	5	lectures/écritures en mémoire centrale (IF, OF, OS, BS)
ALU	6	opérations arithmétiques et logiques (AND, OR, ADD, SUB)
	4	opérations indépendantes de toutes unités (ex : INC (P,P)
DE SEQUENCE	7	contrôle de séquence (GO TO, JUMP, RETURN)
DECALAGE	8	décalage des registres.

Grâce à ce nouveau découpage on s'assure que deux micro-opérations de même type et traitant la même ressource, appartiendront toujours à deux micro-instructions différentes.

Comme nous l'avons déjà signalé plusieurs fois, il existe un problème de séquences à respecter. Ce problème comporte deux aspects. Le premier est résolu par l'intermédiaire des instructions de séquence et des labels (points de branchement) que nous retrouvons dans la table des symboles (ST). L'autre est en fait implicite et est la conséquence de la recherche du parallélisme maximal. Ainsi, par exemple, une opération sur des registres ne peut s'effectuer que lorsqu'ils ont effectivement le contenu désiré par le microprogrammeur . Cette vérification s'exprime aisément par un graphe, mais pour le construire, il faut connaître "qui utilise ou modifie quoi". Ainsi, dans l'exemple de la figure II-6, il est clair que l'opération (3) ne peut s'effectuer correctement qu'après achèvement des opérations (1) et (2) qui modifient les registres 1 et 2 dont elle se sert.

La structure du langage source implique qu'il n'y aura jamais plus de deux ressources modifiées et plus de deux ressources utilisées par micro-opération.

Dans le langage intermédiaire, nous trouvons également un pointeur d'alternative. Une analyse détaillée de la micro-ins-



Contient les labels des micro-opérations de séquence. Celles-ci sont séparées par un zéro.

FIG. II-5 : Structure du langage intermédiaire.

Langage intermédiaire (ILT)

Volet 1

CHAMPS

	TS	AF	MS	MT	FS	TF	SF	GF	MR	AB	IM	LB	LA	RF	FF	MF	CF	WR	SC	VF	WF	XF	SH	BB	AA	ALTER	TYPES	MODIF	UTIL	LABEL
							01				1101															1	5	17	17	
												01			1010	1								0001			6	44	17	
						00	00	1																			1	33		
						00	11	0111																		2	2	42	33	
				0	0000	10																				3	7		42	1
							00				0001																5	17,43		
												01		011	1010	1								0001			6	20	17	
			00101		1101																						7		17	4
												01	00		1010	1		1						0101	1110		6	14	20	
10												11	00	011	1011	0	00	0	0	0	0	01	111	1111	1110		6	20	14	
11	0000			0	0000																						7			7
12												01	00		1010	1		1						0100	1110		6	14	20	
13												11	00	011	1011	0	00	0	0	0	0	01	111	1111	1110		6	20	14	
14												01			1010	1		1						0000	1111		6	15	20	
15												10		011	1010	1	11	1	1	1	1	11	111	1111			6	20		
16												01			1010	1		1						0000	1110		6	14	20	
17													00	011	1111	1									1111		6	20	15	
18							00				0100		00		1011	1	11	1	1	1	1	01	111	1111	1110		6	47	14	
19														100													4	16	16	
20							01				1000															4	5	17,46	16	
21	1000						00			10		01			0001	0								0000			6	48	20,48	
22														100													4	16	16	
23							01				1000															5	5	17,46	16	
24	0000					00	00				0011															6	7			9

```
RD      OF(MIR)
        TRN(MIR, )
        SAMPLE(ALU)
        TEST(ALUS)
        GOTO(RD,TRAIT)
TRAIT   :WAIT
        TRN(MIR,OPR)
        GOTO(IR0,MSK10000,GAUCHE,DROIT)
GAUCHE  :TRN(OLSE,R14)
        AND (R14,X'00FF,OPR)
        JUMP(TRF)
DROIT   :TRN(OLSE,R14)
        AND(R14,X'00FF,OPR)
TRF     :TRN(OPR,R15)
        TRN(MSKFFFF,OPR)
        TRN(OPR,R14)
        TRN(R15,OPR)
        AND(R14,X'FEFF,IR)
        INC(P,P)
        IF(P)
        OR(RS5,OPR,RS5)
        INC(P,P)
        IF(P)
        JUMP(0,SS3M)
```

Volet 2

						10	00				1101																			
						00	11	0111																						
				0	0000	11																								
						10	00				1000																			
						10	00				1000																			
						00	10	1																						

table des symboles ST

NOMS	DEF	REF
DROIT	11	8
GAUCHE	9	8
AD	1	5
SS3M		18
TRAIT	6	5
TRF	12	10

tables des correspondances CT

NSOUR	NLI	NSOUR	NLI
1	1	10	11
2	2	11	12
3	3	12	14
4	4	13	19
5	5	14	20
6	6	15	21
7	7	16	22
8	8	17	23
9	9	18	24

Volet 3

0	TRAIT	0	GAUCHE	DROIT
	TRF	0	SS3M	0

SYMBOLES UTILISES

DEF	N° de la micro-opération de définition du symbole
REF	N° de la micro-opération référencant le symbole
NSOUR	N° de la micro-opération source
NLI	N° de la micro-opération du langage intermédiaire

Figure II-5b : Exemple de traduction du langage source en langage intermédiaire.

truction de la VARIAN fait en effet apparaître une redondance possible d'expression de la même micro-opération de notre langage. Il est intéressant d'exploiter au maximum cette redondance ; c'est pourquoi, comme il est montré dans la figure II-5, le langage intermédiaire est formé de deux volets ; l'un exprimant les champs "standards" d'expression et l'autre exprimant les façons différentes de le faire.

Il reste à définir les champs qui correspondent à la micro-opération et à préparer la composition. De nombreuses contraintes hardware sont exprimées par l'intermédiaire des champs. Ainsi deux micro-opérations utilisant les mêmes unités (ALU-décaleur ...) auront les mêmes champs définis de la même façon. Une comparaison des champs permettra d'éviter le regroupement de ces micro-opérations dans la même micro-instruction. Pour pouvoir réaliser ces comparaisons entre champs, une valeur neutre nous est nécessaire ; c'est pourquoi chaque bit est dédoublé.

La figure II-5 décrit la structure adoptée pour le langage intermédiaire. Elle laisse apparaître deux parties : l'une dépend de la microprogrammation de la VARIAN puisque les champs sont positionnés en fonction de la micro-opération et du chemin des données de cette machine,

- l'autre, affectée uniquement par le code opération (TYPE) et les opérandes (MODIF - UTIL) est, au contraire, indépendante de tout matériel.

Si nous reprenons le modèle général de la figure I-6, nous constatons que la partie indépendante servira à reconnaître les concurrences et à effectuer une première composition de micro-instructions tandis que la partie dépendante ajustera cette composition en fonction du hardware utilisé.

2.2. Du langage source au langage intermédiaire

Pour traduire le langage source en microprogramme, nous aurions pu, comme l'assembleur 360, organiser la table MOT (mnémoniques des opérations) de telle sorte qu'elle fournisse le nom de champs à positionner et leur valeur. Une seconde passe aurait alors traité les opérandes suivant le type de l'opération. Dans notre cas, ce travail ne peut être effectué de manière systématique sans nécessiter des tables énormes. En effet, une même opération positionne des bits différents suivant les opérandes. De plus, nous avons vu qu'une même opération pouvait se traduire de différentes manières.

exemple : La table MOT en assembleur 360

Mnémonique		code opération	
"SRA"	⟶	8A	
"SRDA"	toujours ⟶	8E	traitement des opérandes
"SRDL"	⟶	8C	

correspondrait dans notre langage à

Mnémonique	code opération (= champs à positionner)				
	LA	SC	WF	XF	SH
SHFTR (R1,ARITH)	3	-	-	-	2
SHFTR (OPR,ARITH)	-	1	0	2	-

Nous avons donc choisi une solution software dynamique plutôt que la solution statique des tables. La table MOT donne en fait l'adresse d'une routine de traitement. Elle se présente sous la forme de la table reprise à la figure II-7. Ce sont les différentes routines appelées après la reconnaissance des mnémoniques qui positionnent les champs, créent les alternatives et remplissent les parties MODIF, UTIL et TYPE du langage intermédiaire.

Cette manière de procéder se justifie d'autant plus que, ce travail n'étant qu'un premier pas dans l'élaboration d'un langage de microprogrammation, il importe de s'orienter le plus possible vers une forme modulaire de programmation.

2.3. Fonctionnement détaillé du traducteur

Après une brève analyse de la partie syntaxique (fig.II-2), nous procèderons à un examen plus approfondi des différentes fonctions et des tâches effectuées dans chacune d'elles.

Rappelons que le but de la phase d'analyse syntaxique est d'extraire les micro-opérations du langage source, de prendre un maximum d'informations pour préparer les travaux d'optimalisation et d'adressage qui suivent. Nous commençons par mettre le compteur de micro-opérations à zéro. Ce compteur permettra d'établir la table des symboles et la table des correspondances. Le label éventuel est alors traité : sauvetage dans la table des symboles (ST). Nous recherchons ensuite l'opération à effectuer et son examen détermine s'il s'agit d'une pseudo-opération (POTGET) ou d'un code-opération (MOTGET). La table des micro-opérations (MOT) donne la routine à exécuter (TORi) pour traiter l'opération (fig. II-7). Cette séquence est alors répétée. Cette boucle n'est qu'une petite partie de la phase d'analyse, mais elle en est la plus importante.

La majorité des modules de la figure II-2 sont suffisamment généraux et bien connus pour nous abstenir de les décrire plus en détail. Seuls le module de mise à jour de la table des symboles (STSTO) et les routines de traitement des grands types de micro-opérations (TORi) sont analysés ci-dessous. En outre, nous présentons un exemple qui montre le chemin suivi du langage source au langage intermédiaire

2.3.1. Description de la routine de mise à jour de la table des symboles

Le module STSTO peut être appelé par la routine de recherche des pseudos-opérations (POTGET), la routine des micro-opérations (MOTGET) et enfin exécutée indépendamment. Schémaniquement nous avons :

```
TRN  (X'4,R1)        (1)
TRN  (X'12,R2)       (2)
ADD  (R1,R2,R0)      (3)
SUB  (R0,R1,R3)      (4)
AND  (R1,R2,R4)      (5)
OR   (R0,R4,R4)      (6)
ADD  (R2,R1,R5)      (7)
ADD  (X'3,R2,R6)     (8)
```

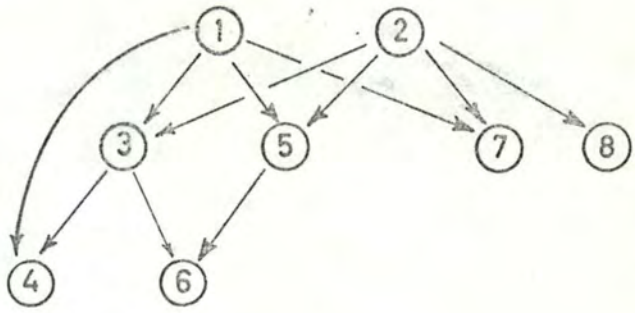
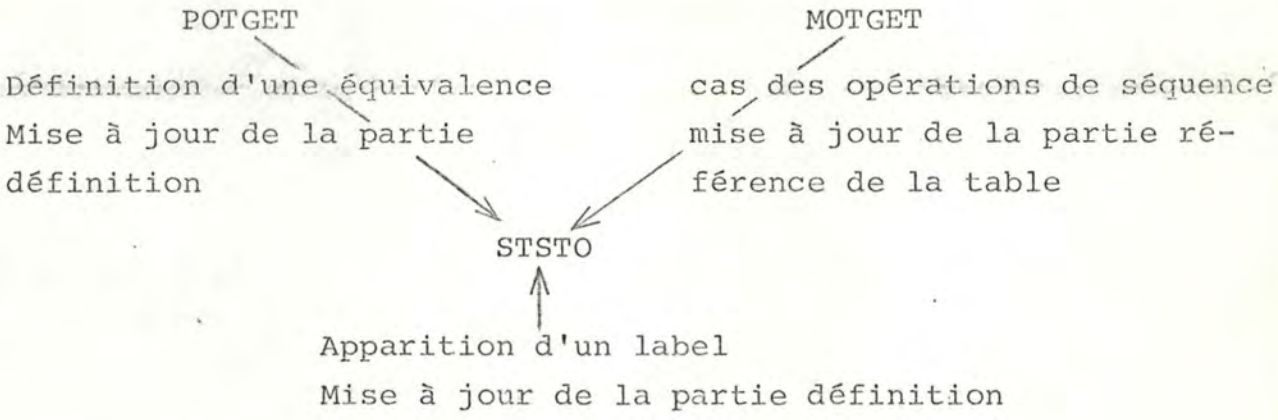


FIG. II-6 : Exemple de séquence "implicite" entre micro-opérations

Mnémoniques	Routines à exécuter
ADD	TORALU
AND	TORALU
BS	TORMEM
CALL	TORSEQ
DEC	TORALU
DECODE	TORSEQ
GOTO	TORSEQ
IF	TORMEM
INC	TORALU
JUMP	TORSEQ
NOT	TORALU
OF	TORMEM
OR	TORALU
RESET	TORTST
RETURN	TORSEQ
SAMPLE	TORTST
SET	TORTST
SHFTL	TORSHFT
SHFTR	TORSHFT
SUB	TORALU
TC	TORALU
TEST	TORTST
TRN	TORALU
WAIT	TORMEM
XOR	TORALU

FIG. II-7 : Description de la table MOT : mnémoniques des micro-opérations

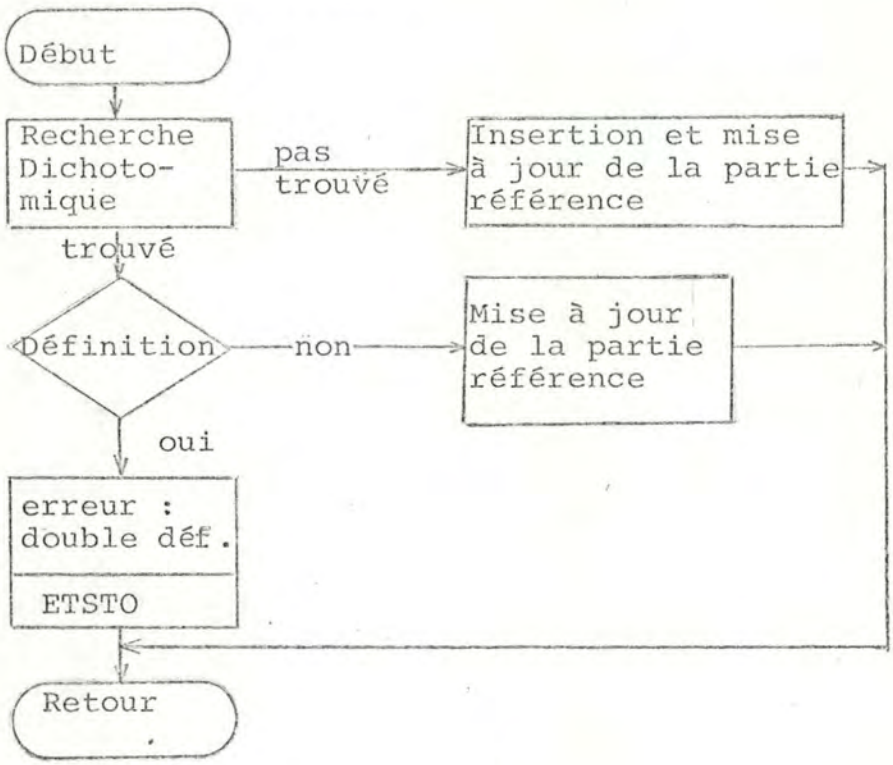


La table des symboles a la structure :

Nom	numéro micro-op. de définition	numéro micro-op. de référence

Nous trions la table à chaque appel de STSTO par un algorithme de recherche dichotomique. Cela permet de systématiser la recherche et l'insertion d'un symbole. Ainsi il n'est pas nécessaire d'avoir des procédures différentes suivant l'origine de l'appel.

STSTO (nom du symbole, définition / référence)



2.3.2. Description des différentes routines de traitement

Nous savons qu'au départ, la table MOT (mnémoniques des opérations) choisit la routine de traitement à exécuter. Dans ce paragraphe, nous donnerons un exemple (TORMEN) de ces routines ainsi que des commentaires généraux permettant de construire les autres sur le même schéma logique. Le but de ces routines est de transformer le langage source en un langage intermédiaire (fig. II-5). Nous avons vu que ce dernier était composé d'une partie dépendante du hardware (parties CHAMP-ALTERNATIVE) et d'une partie utilisée lors du regroupement des micro-opérations sur base de l'utilisation des ressources de ces dernières. Enfin, c'est aussi aux routines TORi qu'il incombe d'éclater les micro-opérations fournies à l'utilisateur, mais intraduisibles directement en terme de bits positionnés dans les champs de la micro-instruction. Ainsi, par exemple, nous permettons un travail en mémoire sur une adresse fournie par un registre général (ex : IF (R3)), celui-ci n'étant pas connecté au registre adresse mémoire, MAD (voir schéma des connexions en annexe 1). Il faudra dès lors éclater cette opération en opérations réalisables (dans l'exemple : TRN (R3,); IF (ALU)). Dans le langage créé, de telles micro-opérations ne nécessitent jamais de sauver le contenu de ressources pour les rendre réalisalbes. Un tel sauvetage de ressources impliquerait rapidement une gestion de toutes les ressources et diminuerait considérablement l'optimalité du code généré.

La figure II-8 présente TORMEM sous forme de table de séquence. Celle-ci est facilement compréhensible à la lumière de ce qui précède.

2.3.3. Exemple de traduction

Le travail de l'analyseur syntaxique est illustré sur un microprogramme plaçant un byte de la mémoire dans les 8 bits de poids faibles d'un registre R0 → R7). La référence au microprogramme à partir de l'assembleur se fera par un "branch to control store":

BCS j, reg, IC

où "j" spécifie de quel byte du mot il s'agit (j = 0 byte droit ; j = 1 byte gauche) "reg" donne le numéro du registre (0 à 7). "IC" sera l'adresse de la WCS où le microprogramme débute. Cette instruction sera suivie par l'adresse du mot contenant le byte, celle-ci pouvant être directe (i = 0) ou indirecte (i = 1).
Nous avons donc :

	BCS			
1	1 1 0 1 000	j	reg	A(IC)
2	i	adresse		

Logiquement, il faut d'abord une boucle pour aller lire le byte en mémoire (opérations 1 à 5). Celui-ci trouvé, nous le cadrons dans l' OPR, nous préparons le registre concerné afin de conserver le byte de gauche et nous opérons l'insertion du byte. A cette logique de base, nous pouvons ajouter quelques opérations (14, 15, 17, 18) pour conserver le "pipe line" dont on bénéficie au début du programme.

COMMENTAIRES	PHASES	IF	OF	OS	BS	WAIT	OVR	P	ALU	MIR	Rn	CONDI.	≠	TRAITEMENTS
Choix des opérations	1	2	3	4	5	6								
Opérations : IF	2						7	8	9	10	11			IM = XXOO MODIF = IBR, MIR
OF	3						7	8	9	10	11			IM = XX01 MODIF = MIR
OS	4						7	8	9	10	11			MODIF = MEMOIRE UTIL = ALU IM = XX10
BS	5						7	8	9	10	11			MODIF = MEMOIRE UTIL = ALU IM = XX11
WAIT	6												14	IM = 0001 SF = 0 , TF = 00 MODIF = MIR
Opérandes : OVR	7											12	13	IM = IM U (00XX) UTIL = OUR
P	8											12	13	IM = IM U (10XX) UTIL = P
ALU	9											12	13	IM = IM U (01XX) UTIL = MIR
MIR	10											12	13	UTIL = MIR IM = IM U (11XX)
Rn	11											12	13	Générer TRN (Rn,) IF(ALU)
Opérations conditionnelles :	12													SF = 3 UTIL = test unit Créer 3 alternatives 1) SF=2 TF=1 2) SF=2 TF=2 3) SF=2 TF=3
	13												14	SF = 1 Créer une alternative SF=2 TF=0
	14													Instruction suivante

X : valeur neutre

FIG. II - 8 : Routine TORMEN

	Labels	micro-instructions	commentaires
(1)	RD :	OF (MIR)	Le registre d'entré mémoire donne l'adresse du byte
(2)		TRN (MIR,)	
(3)		SAMPLE (ALU)	
(4)		TEST (ALUS)	L'adressage est-il direct ?
(5)		GOTO (RD, TRAIT)	Si oui on est en train de lire le byte (TRAIT), sinon on recommence la boucle (RD)
(6)	TRAIT :	WAIT	Synchronisation avec l'opération mémoire.
(7)		TRN (MIR,OPR)	Transfert du mot contenant le byte dans l'OPR et branchement
(8)		GOTO IR9,MSK10000, GAUCHE,DROIT)	suivant le byte à extraire
(9)	GAUCHE :	AND (OLSE,X'OOFF, OPR)	
(10)		JUMP(TRF)	cadrage du byte à transférer dans le byte droit de l'OPR
(11)	DROIT :	AND (ORSE,X'OOFF, OPR)	et mise à zéro du byte gauche
(12)	TRF :	AND (MSKFFFF,X'FEFF, IR)	
(13)		INC (P,P)	rétablissement du "pipe-line"
(14)		IF (P)	
(15)		OR(RS5,OPR,RS5)	insertion du byte dans le registre spécifié
(16)		INC (P,P)	rétablissement du "pipe-line"
(17)		IF (P)	
(18)		JUMP (O, SS3M)	prise en charge de l'instruction suivante.

L'analyseur syntaxique prend en charge la première micro-opération, place le symbole RD dans la table des symboles (ST) et la valeur du compteur de micro-opération source dans la partie définition. Il traduit ensuite le "OF(MIR)", à l'aide de la table de séquence de la figure II-8 (TORMEM). Celle-ci traduit une instruction du langage source en une instruction du langage intermédiaire (ILT). La phase 3, puis la phase 10 positionnent le champ IM, à la valeur hexadécimale D, ainsi que les parties MODIF et UTIL. L'instruction n'étant pas conditionnelle, la phase 13 complète le champ SF et crée une alternative, c'est-à-dire que la colonne ALTER est complétée et qu'une entrée est créée dans le volet 2 du langage intermédiaire. Excepté les champs SF et TF, cette entrée est identique à celle construite ci-dessus.

Ce travail est d'une logique relativement simple et peut être appliqué à toutes les micro-opérations du langage source. Les résultats finaux sont explicités à la figure II-5(b).

Chapitre 3

La Composition

3.1. Buts de la composition

A ce stade, les microprogrammes ne sont constitués que de micro-ordres exprimés séquentiellement. Or une machine à format de micro-instruction horizontal permet un certain parallélisme qu'un microprogrammeur exploiterait. Pour être efficient, le compilateur doit également pouvoir l'exploiter. Chercher à regrouper des micro-opérations exige une analyse des ressources utilisées par les micro-opérations et une analyse de la séquence à respecter. Comme la composition doit se faire de façon optimale, une étude des algorithmes d'optimalisation est d'abord effectuée et il n'est pas étonnant que le terme "optimalisation" remplace dans le texte celui de "composition".

L'optimalisation peut se faire à deux niveaux : global et local. L'optimalisation globale travaille sur l'ensemble des instructions. Gries (GRIES, 1971), Allen (ALLEN, 1970), Aho (AHO, 1970) et d'autres analysent le programme dans son ensemble pour en extraire des relations intéressantes au niveau global. Elles serviront dans la suite, par exemple, à éliminer les évaluations de constantes à l'intérieur de boucles. Ces algorithmes généraux peuvent être appliqués ici. Le niveau d'optimalisation global minimise généralement le temps d'exécution des microprogrammes par l'exploitation du parallélisme. Ce qui implique la détection de flots parallèles et le transport de micro-opérations. Pour détecter de tels flots, un microprogramme est partitionné en segments (= blocs) exécutables indépendamment les uns des autres et pouvant être confiés à des processeurs multiples (BERNSTEIN, 1966). L'optimalisation des microprogrammes par transport de code s'effectue sur les segments et détecte les micro-opérations pouvant être transférées dans un autre segment afin de diminuer le temps d'exécution sans affecter le résultat. Si par exemple, une micro-instruction d'une boucle est exécutée avec des opérandes constants, elle peut être transportée en dehors de la boucle et diminuer ainsi le temps nécessaire à l'exécution.

L'optimalisation locale est exécutée sur chaque segment du microprogramme. A ce niveau, la notion de bloc (ou segment) doit être précisée : c'est une portion de microprogramme ayant au plus une instruction de contrôle de séquence vers les autres blocs. C'est l'équivalent d'un noeud dans le graphe réduit des tâches d'un programme dans le sens de Ramamoorthy et Gonzales (RAMAMOORTHY et al., 1969). Le bloc peut être considéré comme indépendant et être analysé sépérément pour la détection du parallélisme possible. L'optimalisation du temps d'exécution s'en déduit vu l'hypothèse faite sur le type de micro-instruction traité (à format horizontal). En effet, la recherche d'une composition minimale ou d'un temps d'exécution minimum conduit à l'exploitation maximale du parallélisme. L'optimalisation locale est rendue difficile par le nombre limité des ressources contrôlables par les micro-instructions. Puisqu'à ce niveau, les opérations parallèles sont exécutées par un processeur unique, cela requiert la détection d'opérations exécutables en parallèle dans le microprogramme et l'allocation à des ressources concurrentes. Deux

opérations exécutables parallèlement dans un microprogramme, par exemple, ne peuvent pas être exécutées simultanément si elles utilisent la même unité (ALU, bus, ...). Les conflits dans l'utilisation des ressources doivent être résolus pour l'optimalisation locale. C'est une des premières difficultés en microprogrammation horizontale. Pour détecter et résoudre automatiquement de tels conflits, un schéma d'allocations de ressources doit être incorporé dans une technique de détection de parallélisme.

3.2. Techniques d'optimalisation existantes

Seules les techniques d'optimalisation utilisant la recherche du parallélisme seront exposées. La littérature décrit suffisamment (GRIES, 1971 ; ALLEN, 1970 ; AHO, 1970) les méthodes travaillant au niveau global.

Dans l'optimalisation d'un programme machine quelconque, le souci principal est de préserver l'algorithme programmé. Ici, le seoond souci est de générer une séquence de micro-instructions minimale à l'intérieur d'un segment. Les algorithmes doivent donc garantir que l'ensemble des micro-instructions de chaque segment soit valide et minimum.

Pour un bloc de n micro-opérations, la technique qui consiste à énumérer toutes les micro-instructions afin de déterminer les conflits de ressource peut demander plus de (n-1) ! comparaisons entre micro-opérations. Atstopas et Plûkas (ATSTOPAS, 1971) garantissent un ensemble de micro-instructions optimal, mais leur stratégie consiste à énumérer tous les ensembles de micro-instructions par bloc et choisissent l'optimum. Yau, Schowe et Tsuchiya (YAU et al., 1974) ont affiné la méthode en remarquant que le nombre de comparaisons demandées peut être réduit si toutes les micro-instructions ne doivent pas être générées et/ou si tous les ensembles ne doivent pas être construits. Le nombre minimum de pas est atteint si seul l'ensemble ayant le minimum de micro-opération est généré. Les micro-opérations sont assignées aux micro-instructions si et seulement si les conflits de ressources et de dépendance des données le permettent. Si une telle méthode garantissait l'optimum, elle serait beaucoup mieux qu'une méthode énumérative. En fait, les auteurs n'arrivent qu'à diminuer le nombre de pas. Leur algorithme ressemble fort à un "Branch and Bound" où à chaque pas un choix est fait et où l'alternative est entièrement mémorisée pour une reprise ultérieure éventuelle. Ramamoorthy et Tsuchiya (RAMAMOORTHY, 1974) ont créé un langage de micro-programmation horizontal (SIMPL) et l'utilisent avec un compilateur optimisateur. Pour chaque bloc, ils déterminent la date au plus tôt et la date au plus tard d'exécution de chaque micro-opération. Ils obtiennnent alors des micro-opérations critiques (date au plus tôt = date au plus tard). Ils assignent ensuite ces micro-opérations à des micro-instructions. Les micro-opérations non critiques sont ordonnées suivant leur date au plus tôt d'exécution et leur séquence. Elles sont finalement comparées avec les micro-instructions précédemment créées pour y être incluses.

3.3. Options choisies

Précisons d'abord que l'optimalisation globale n'est pas prise en considération. D'une part, elle est très longue et n'apporterait probablement pas de grands résultats vu la simplicité de notre langage. De plus tout bon programmeur sait, par exemple, que les calculs de constantes sont à placer hors des boucles. Enfin, si l'optimalisation globale était retenue, elle pourrait trouver place dans une première phase de l'analyse.

Pour ce qui est de l'optimalisation locale, nous avons repris l'idée de Ramamoorthy & Tsuchiya(RAMAMOORTHY, 1974) qui consiste à partir des principes de 'assignation unique que Tesler et Enea (TESLER, 1968) ont développé dans COMPEL (compute parallel), un langage de haut-niveau pour le multiprocessing. Un programme écrit dans un langage d'assignation unique à les caractéristiques suivantes :

+ Aucune variable n'est modifiée par plus d'une instruction.

  Le seul effet de l'exécution d'une instruction est d'assigner des valeurs à certaines variables. Elle n'affecte jamais le contenu des variables d'entrée (pas de "side effect").Toutes les instructions sont des assignations. Les variables appartiennent à un des groupes :

  - sortie : celles qui reçoivent une valeur ;
  - entrée : celles dont la valeur est utilisée dans l'instruction.

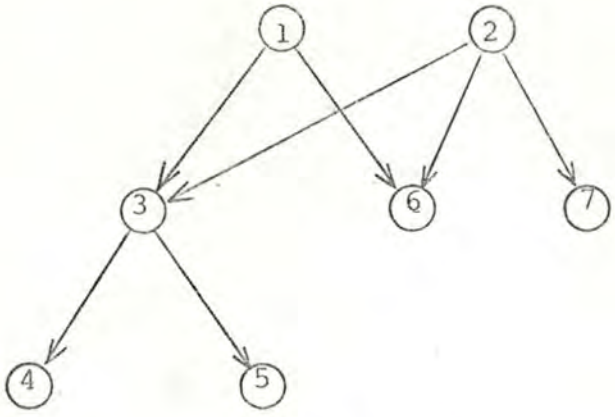
+ L'instruction donnant une valeur à une variable doit s'exécuter avant toute instruction l'utilisant en entrée ou utilisant une valeur qui en dépend.

Une façon d'optimaliser un tel programme est de réduire le nombre de calculs redondants par une combinaison des expressions communes. Dans un programme d'assignation unique il n'y a pas de "side effect", donc les expressions communes du programme peuvent être combinées pendant la phase de compilation. Un autre moyen serait l'allocation efficiente des mémoires. Les restrictions données dans un tel langage permettent qu'une instruction soit exécutée sans tenir compte de la séquence dès que toutes ses variables d'entrée ont une valeur.

Exemple de programme d'assignation unique

```
B = 4       (1)
C = 12,5    (2)
A = B/C     (3)
D = A*B     (4)
E = C↑B-A   (5)
F = C/B     (6)
G : 3*C     (7)
```

Les instructions (1) et (2) peuvent s'exécuter directement et simultanément puisqu'elles ne dépendent d'aucune variable. Après que B et C aient reçu une valeur, (1) et (2), toutes les instructions ne nécessitant qu'une de ces deux variables (ou les deux), (3), (6) et (7) peuvent être exécutées simultanément. Et ainsi de suite ... D'où le graphe de dépendance :



Les propriétés de l'assignation unique permettent donc la détection des instructions exécutables simultanément. Ces concepts ont été approfondis par Chamberlin (CHAMBERLIN, 1971) dans la création du SAMPLE (Single Assignment Mathematical Programming Language).

Pour que ces concepts soient applicables à la microprogrammation, il nous faut ajuster les différentes définitions de l'assignation unique. D'abord l'environnement d'application est assez différent. Nous n'avons pas un multi-processeur, mais un ensemble multiple d'opérations élémentaires. La difficulté provient des différences suivantes :

1. Les variables des microprogrammes représentent des ressources mémoires (registres-indicateurs) ;
2. ce n'est pas aux ressources, mais plutôt à leur contenu, que les concepts de l'assignation unique doivent être appliqués ;
3. des processus parallèles sont réalisés en microprogrammation par des chemins de données parallèles (bus) et des registres plutôt que par des processus multiples;
4. des processus parallèles sont exécutés par des unités multiples asynchrones tandis qu'ici les micro-opérations sont exécutées de manière synchrone durant un temps d'horloge.

Pour appliquer les principes de l'assignation unique, l'ensemble des micro-opérations est partitionné en blocs. Les blocs étant définis, une analyse du parallélisme est exécutée sur chacun d'eux. Pour cela, il faut encore préciser qu'une variable ayant reçu une valeur est considérée comme définie pour toutes les instructions apparaissant avant une instruction lui assignant une nouvelle valeur. Ainsi dans l'exemple :

$X \rightarrow A \quad (1)$
$Q \rightarrow B \quad (2)$
$A \wedge Q \rightarrow Q \quad (3)$
$X+1 \rightarrow A \quad (4)$
$X-B \rightarrow P \quad (5)$

Les instructions (1) et (4) nedépendant que de X pourraient s'exécuter simultanément. Cependant l'instruction (3) réfère la valeur définie à l'instruction (1). Il faut donc "retarder" l'exécution de (4) jusqu'àprès l'exécution de (3).

3.4. Algorithmes

Avant de commencer une discussion sur la façon d'obtenir un microprogramme optimal, la définition de quelques termes est nécessaire

Conflit de ressource (entre deux micro-opérations) : si deux micro-opérations théoriquement concurrentes ne peuvent s'exécuter simultanément parce qu'elles utilisent les mêmes ressources de la machine (ne pouvant être partagées), nous disons qu'un conflit de ressource existe entre-elles.

La liste Rj disponible au pas j est la liste des micro-opérations exécutables de façon concurrente à la date j s'il n'y avait pas de limite hardware, c'est-à-dire si la machine avait des ressources de chaque type en nombre infini.

Une micro-instruction complète est une micro-instruction qui, du fait des limites du hardware et/ou de son encodage ne peut plus accepter aucune des micro-opérations de Ri.

Les micro-opérations libres sont celles n'appartenant pas au chemin critique c'est-à-dire celles dont les dates d'exécution au plus tot et au plus tard forment un intervalle de temps.

Un graphe de dépendance est un graphe connexe et acyclique qui représente la dépendance des données pour toutes les micro-opérations du bloc. Chaque noeud représente une micro-opération du bloc et chacune d'entre-elles est représentée par un et un seul noeud. Les arcs du graphe expriment la dépendance des données qui existent entre les micro-opérations. Chaque noeud est directement dépendant des données de ses prédécesseurs immédiats et indépendant des données de ceux qui ne sont ni ses prédécesseurs ni ses successeurs.

Comme pour la partie précédente, ce paragraphe décrit les fonctions principales composant la phase. Dans la figure II-3, un découpage du programme en blocs est effectué. Un bloc étant constitué par une séquence d'instructions terminée par une opération de contrôle de séquence, il est facilement identifiable à partir de la table des symboles qui fournit les numéros des micro-opérations définissant les labels et celles les référenciant. Ces numéros sont traduits en numéros de micro-opérations du langage intermédiaire afin d'aller les rechercher lors de l'établissement du graphe de dépendance. Tout cela ne constitue en fait que la préparation des données au travail de composition proprement dit.

Le langage intermédiaire partitionné en blocs est analysé pour détecter le parallélisme local. L'analyseur de concurrences examine chaque bloc pour trouver les micro-opérations exécutables simultanément et déterminer les moments d'exécutions possibles. En particulier, cette analyse détermine, sans tenir compte des limitations hardware, le nombre minimal de cycles nécessaires à l'exécution de chaque bloc. Elle travaille en quatre étapes :

La 1ère étape balaye séquentiellement chaque bloc, détermine la dépendance des variables et établit le graphe de dépendance. Celui-ci constitue le parallélisme maximal possible en

supprimant les conflits de ressources du microprogramme (fig.II-14). En appliquant les principes de l'assignation unique, la 2ème étape détermine la date d'exécution au plus tôt de chaque micro-opération (fig. II-15).

La construction du graphe de dépendance nécessite la tenue à jour de l'utilisation des ressources. Comme signalé, c'est le but des deux tableaux TABUTIL et TABMOTIF. Ils donnent en regard de la ressource le numéro de la dernière micro-opération l'utilisant ou la modifiant respectivement. Comment TABUTIL et TABMODIF sont-ils utilisés pour construire le graphe de dépendance ? Cela revient à adapter l'approche de l'assignation unique (voir II-3.3.) à la microprogrammation. Dans l'analyse des ressources utilisées, ce qui est important de connaître, c'est le numéro de la dernière micro-opération qui la modifie. L'utilisation antérieure n'a plus d'importance. Inversément, lors de la détection d'une modification, seule la dernière utilisation est importante, car la modification a déjà été traitée.

exemple :

```
        TRN (R3, R6)        (1)
         =
        ADD (R1, R2, R3)    (4)
         =
        TRN (R6, R3)        (7)
```

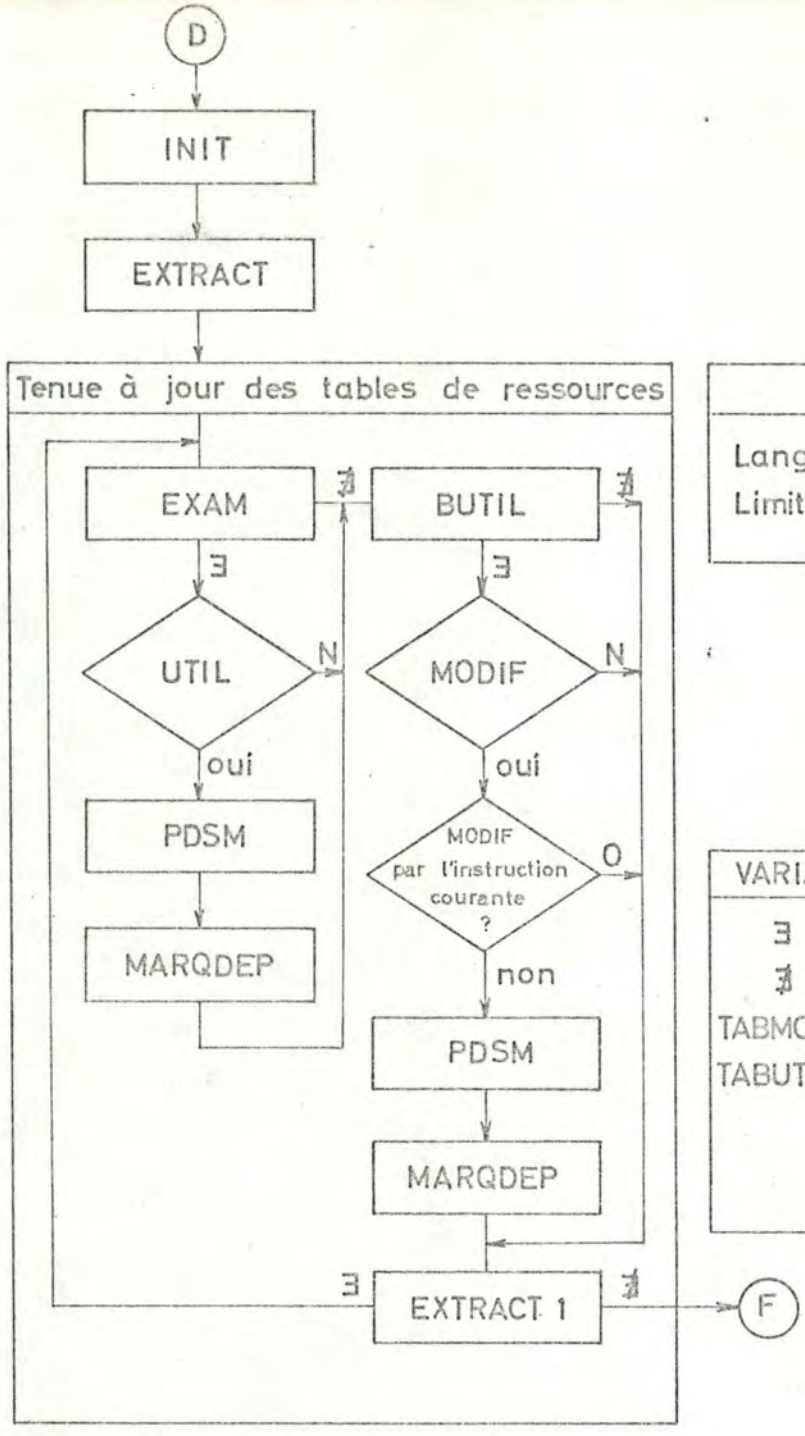
	UTIL	MODIF
R1	4	
R2	4	
R3	1	4 → arc (1) → (4) 7 → arc (1) → (7)
R5		
R6	7	1 → arc (1) → (7)

Le sens de l'arc va toujours de l'ancienne micro-opération vers celle en cours de traitement. Enfin le poids de l'arc vaut 0 ou 1 suivant le cas. Le tableau de la figure II-14 (b) donne les ressources utilisées et modifiées ainsi que le poids de l'arc créé par type de micro-opération. Les poids nuls sont introduits afin d'obliger certaines micro-opérations à être de la même époque que d'autres et donc d'appartenir à la même micro-instruction. Dans ce travail, des unités fictives (ex :test) sont utilisées afin de généraliser le traitement.

La 3ème étape considère l'exécution inverse du microcode du bloc pour trouver la date au plus tard de leur exécution.

La 4ème étape balaye les micro-opérations ayant leur moments d'exécution au plus tôt et au plus tard égaux. Puisque de telles micro-opérations sont critiques elles seront rattachées à des micro-instructions critiques.

Le parallélisme détecté dans cette phase de compilation étant le parallélisme maximal, on pourrait prouver théoriquement (principe de Bellman R., "Une politique est optimale si à une période donnée, quelles que soient les décisions précédentes, les décisions qui restent à prendre constituent une politique optimale en regard du résultat des décisions précédentes") que les micro-opérations critiques constituent une séquence minimum pour un microprogramme complet disposant de ressources illimitées. Ces micro-opérations seront utilisées dans une analyse suivante pour générer un ensemble minimum de micro-instructions pour un bloc.



PARAMETRES
Langage intermédiaire
Limites du bloc (= N° micro-opérations)

VARIABLES ET SYMBOLES UTILISES	
∃	Exprime l'existence
∄	Exprime la non existence
TABMODIF	Table de tenue à jour de la
TABUTIL	modification (de l'utilisation) des ressources par les micro-opérations.

FONCTIONS DES MODULES	
INIT	Initialisation des tables de ressources (TABMODIF et TABUTIL)
EXTRACT	Prise en charge première micro-operation du bloc (soit X)
EXAM	Identification des ressources modifiées par la micro-opération
UTIL	La ressource est-elle utilisée auparavant ?
PDSM	Recherche du poids de l'arc X⟶Y, Y etant le n° de la micro-opération utilisatrice
MARQDEP	Mise à jour du graphe de dépendance
BUTIL	Identification des ressources utilisées par la micro-opération
MODIF	La ressource est-elle modifiée auparavant ?
EXTRACT 1	Prise en charge de la micro-opération suivante et détection de la fin du bloc

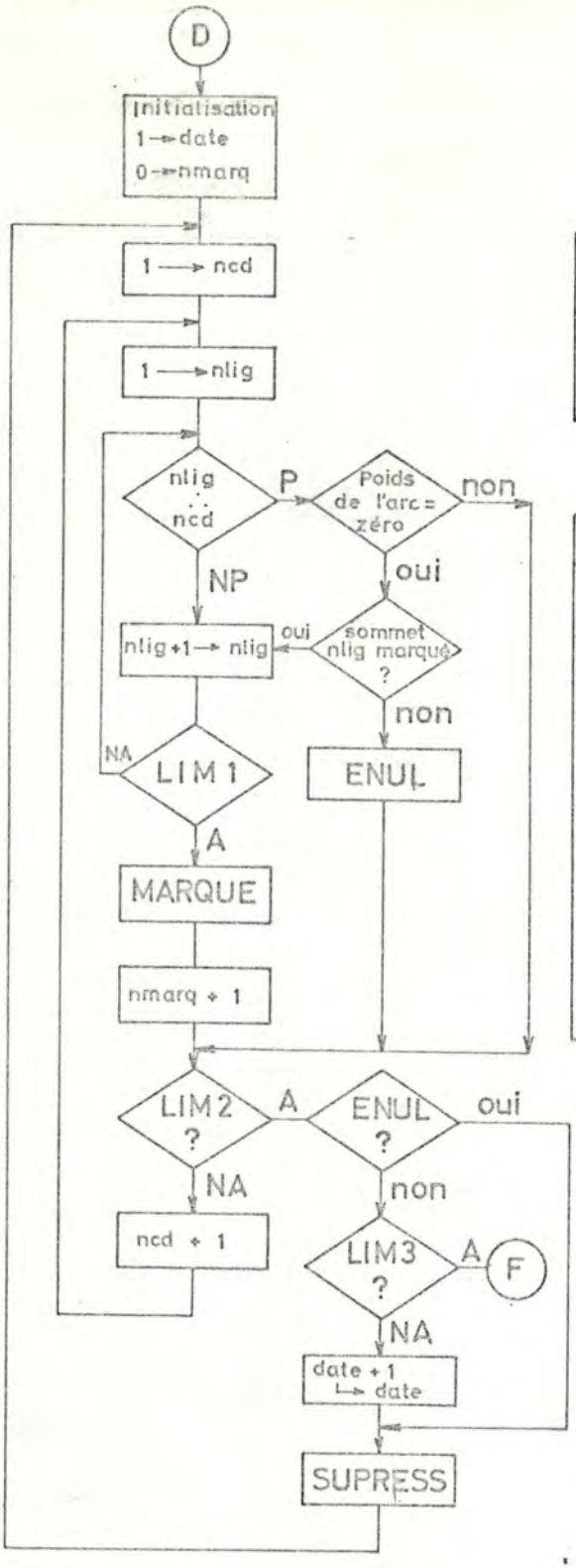
FIG. II-14 (a) : DGSTO, établissement du graphe de dépendance

Opérations	Ressources		Poids
	utilisées	modifiées	
Avec l'ALU			
Op (opl, op2, dest)	test opl, op2	dest, ALU	1
Avec la mémoire			
IF (opl)	opl, mem	MIR, IBR	1
OF (opl)	opl, mem	MIR	1
Op (opl, test)	opl, mem (1) test	MIR, mem (2)	0 (3)
Test			
SAMPLE (flag)	ALU	flag	0
TEST (flag)	flag	test	1
SET (flag)		flag	1
RESET (flag)			
De séquence			
GO TO (vrai, faux)	test		0
GO TO (IR, ...)	IR		1

(1) dans le cas d'un IF ou d'un OF
(2) dans le cas d'un OS ou d'un BS
(3) l'arc issu de l'opération de test à un poids nul, tous les autres ont un poids unitaire.

Symboles utilisés	
mem	unité mémoire
ALU	unité arithmétique et logique
test	unité de test
Op	code opération
op	opérande

Figure II-14 (b) : Tableau résumé des poids des arcs et des ressources utilisées ou modifiées par les différentes micro-opérations.



PARAMETRES
Graphe de dépendance
Nombre de sommet du bloc:nsom

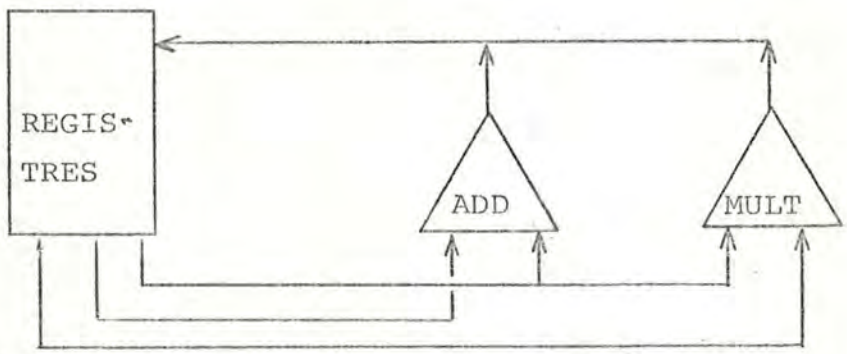
VARIABLES ET SYMBOLES UTILISES	
date	n° de la période en cours d'examen
nmarq	nombre de sommet marqué
ncd	n° du sommet destination (suivant)
nlig	n° du sommet origine (précédent)
P	précédence vérifiée
NP	pas de précédence détectée
A	limite atteinte
NA	limite non atteinte
∴	comparaison de deux éléments

FONCTIONS DES MODULES	
LIM 1	Tous les précédents potentiels (nlig) ont-ils été analysés ?
MARQUE	Marquer le sommet avec la "date" en cours
LIM 2	Tous les sommets (ncd) ont-ils été analysés ?
ENUL ?	Existe-t-il des sommets reliés par des arcs de poids nul ?
LIM 3	Tous les sommets sont-ils marqués (nmarq = nsom) ?
SUPRESS	Supprimer du graphe les sommets marqués
ENUL	Signaler qu'il existe des arcs de poids nuls dans le cycle en cours

FIG. II-15 : Recherche de la date d'exécution au plus tôt

La phase suivante introduit les contraintes de la machine. L'organisation hardware et les caractéristiques des opérations possibles sont définies par la représentation de la micro-instruction. La liste des micro-opérations critiques générées dans la phase précédente est balayée pour construire le flot de contrôle principal du bloc. L'utilisation des ressources des micro-opérations critiques ayant le même moment d'exécution est examinée pour trouver les conflits éventuels et créer une séquence de micro-instructions résolvant ces conflits. Ainsi deux micro-opérations critiques appartenant au même moment d'exécution et qui utilisent la même ressource (ex : ALU) ne peuvent s'exécuter simultanément, deux micro-instructions seront donc nécessaires pour exprimer un tel cycle. Si par contre, il n'y a ni conflit de ressource ni conflit d'encodage, elles seront groupées pour être exécutées simultanément. Ce processus génère un ensemble minimum de micro-instructions sans conflit de ressource incluant toutes les micro-opérations critiques. Les micro-opérations libres restantes sont alors insérées dans cet ensemble de micro-instructions compte tenu de leur temps d'exécution. Les procédures générales de détection de conflits de ressources entre micro-opérations critiques et d'insertion des micro-opérations libres sont illustrées respectivement par les figures II-16 et II-17. La figure II-16 montre la procédure de réarrangement des micro-opérations critiques en tenant compte des contraintes hardware représentées dans notre cas par le fait que deux micro-opérations de même type ne pourront pas être placées dans la même micro-instruction. A ce stade, les conflits d'utilisation des ressources ont tous été pris en considération par le graphe de dépendance des ressources et par le type de la micro-opération. Mais il faut remarquer que dans certaines machines ayant, par exemple, un additionneur et une unité de multiplication distincts des conflits dans l'utilisation des registres pourraient encore se produire à ce niveau du fait de l'utilisation d'un bus commun .

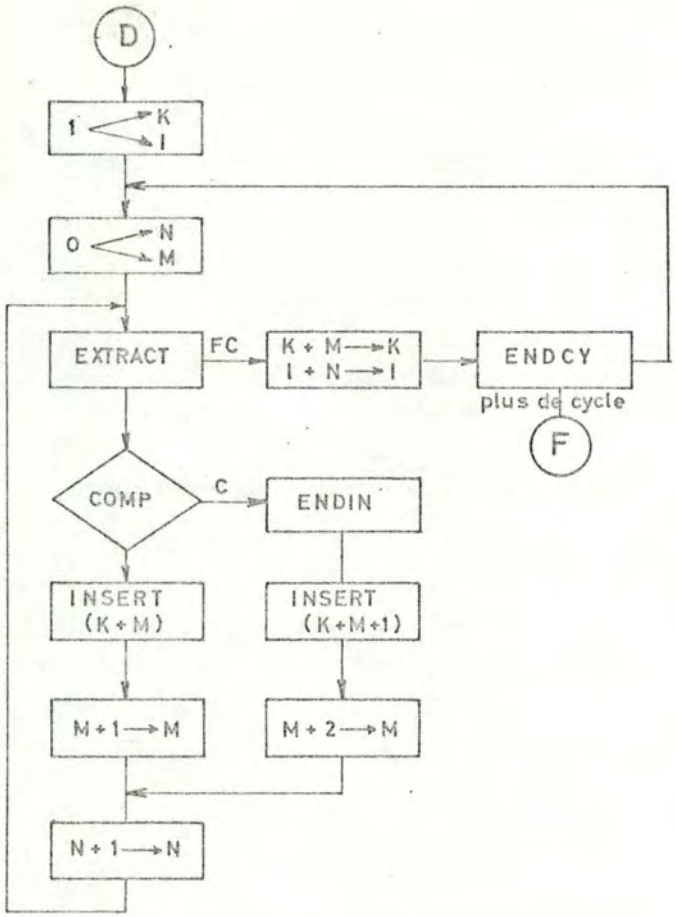
Signalons enfin qu'un conflit de ressources entre deux micro-opérations est exprimé par le positionnement de champs identiques à des valeurs différentes dans les deux micro-opérations.



Exemple :

$R_1 + R_2 \rightarrow R_3$
$R_1 + R_4 \rightarrow R_5$
} pourraient s'exécuter simultanément

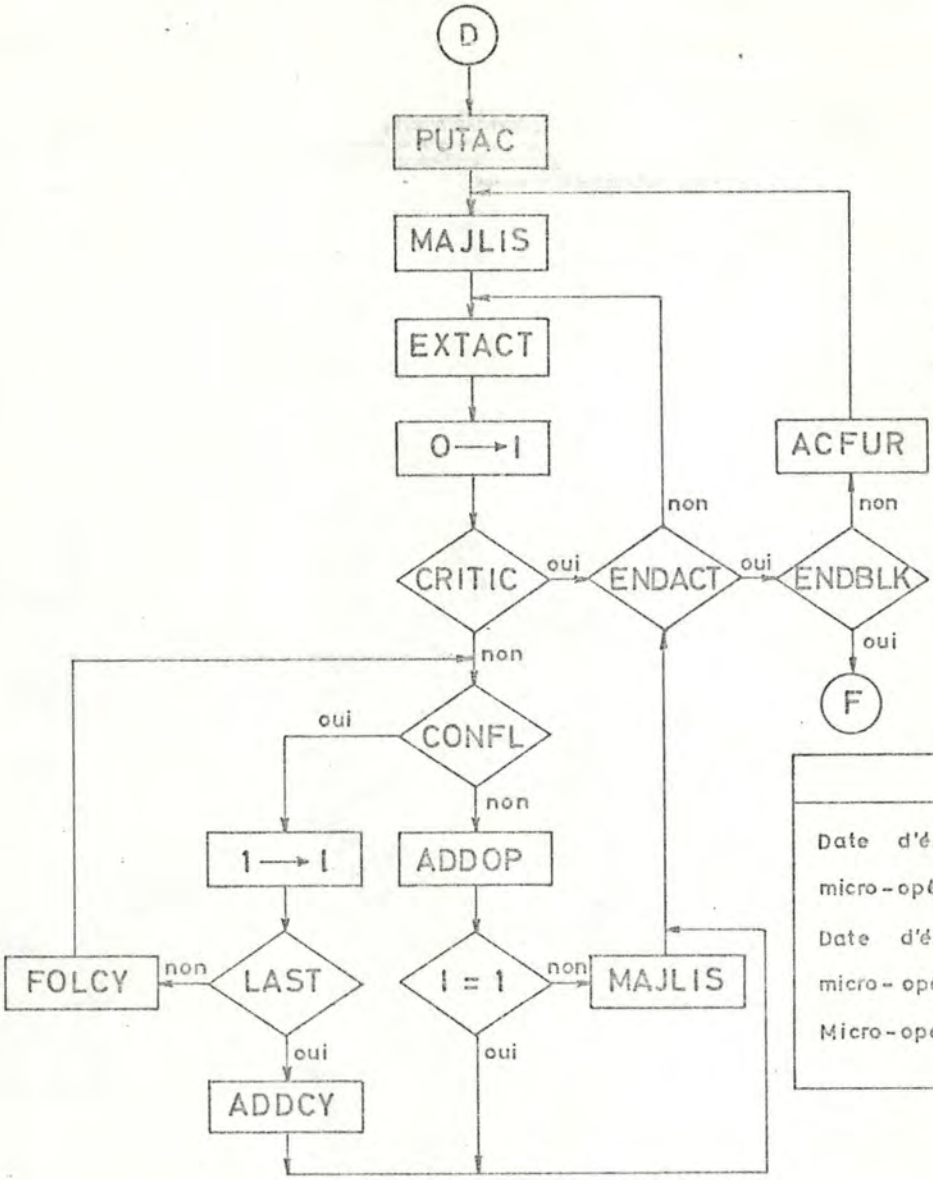
$R_1 + R_2 \rightarrow R_3$
$R_4 * R_4 \rightarrow R_5$
} ne peuvent pas s'exécuter simultanément



PARAMETRES	
Micro-opérations par cycle DACRIT	
VARIABLES ET SYMBOLES UTILISES	
I	n° du cycle en cours d'analyse
K	n° du cycle en cours d'élaboration
N	n° de la micro-opération dans le cycle I
M	n° de la micro-opération dans le cycle K
FC	fin de cycle
C	conflit détecté

FONCTIONS DES MODULES	
EXTRACT	Prise en charge de la micro-opération I de DACRIT
COMP	Comparer la micro-opération I avec les micro-opérations du cycle K de DACRIT 1 pour détection des conflits de type
INSERT	Ajouter la micro-opération au cycle K de DACRIT 1
ENDIN	Placer "fin de micro-instruction" dans DACRIT 1 (K + M)
ENDCY	Indiquer fin de cycle et prendre le cycle suivant en compte

FIG. II-16: Résolution des conflits entre micro-opérations critiques



Les listes ACTUEL et FUTUR contiennent respectivement les numéros des micro-opérations pouvant s'éxécuter au cycle examiné et au cycle suivant.

PARAMETRES
Date d'éxécution au plus tôt des micro-opérations
Date d'éxécution au plus tard des micro-opérations
Micro-opérations critiques : DACRIT 1

FONCTIONS DES MODULES	
PUTAC	Placer les instructions éxécutables au premier cycle dans ACTUEL
MAJLIS	Mise à jour des listes ACTUEL et FUTUR
EXTACT	Prise en charge de la micro-opération suivante de ACTUEL
CRITIC	L'opération I appartient-elle au chemin critique ?
CONFL	Détection des conflits avec les opérations de la micro-instruction examinée
ADDOP	Ajouter la micro-opération au cycle examiné
LAST	Le cycle est-il le dernier possible pour la micro-opération ?
ADDCY	Insertion d'un nouveau cycle et placement de la micro-opération
FOLCY	Prise en charge du cycle suivant pour examen de ses micro-opérations
ENDACT	La liste ACTUEL est-elle terminée ?
ENDBLK	Le bloc est-il terminé ?
ACFUR	Transfert de la liste FUTUR dans la liste ACTUEL

FIG. II - 17 : Procédure d'insertion des micro-opérations libres

Cet algorithme compose les micro-opérations mais dans certains cas, cette composition peut ne pas être la meilleure. En effet, lors de l'insertion des micro-opérations dans les micro-instructions critiques, seule une combinaison est analysée et choisie. Le choix peut ne pas être le meilleur et trop de micro-instructions sont alors générées. Ainsi dans l'exemple suivant l'algorithme génére trois micro-instructions quand deux suffisent.

```
Exemple : soit le bloc : A    INC (P, P)
                         B    AND (RO, X'OFOF, R1)
                         C    ADD (R2,MIR, R2)
                         D    IF (P)
                         E    SHFTL (OPR, ARITH)
                         F    TEST (ALUS)
                         G    GO TO (TRUE, FALSE)
```

L'établissement du graphe de dépendance montre que deux micro-opérations sont critiques (A et D) et les conflits se situent entre B et C, D et C, B et E. L'algorithme d'insertion crée alors trois micro-instructions ;

```
cycle 1 : A, B     (1)
          C, E     (2)
cycle 2 : D, F, G  (3)
```

alors qu'en réalité deux micro-instructions pouvaient suffire: A C E et D B F G.

En pratique de tels cas sont rares et l'algorithme est satisfaisant. Cependant, pour des programmes demandant un minimum absolu, un algorithme conduisant au nombre minimum de micro-instructions par bloc est donné à la fin du chapitre. Cet algorithme, fournit l'optimum mais au prix d'un temps de compilation très long et d'un espace mémoire conséquent. Son utilisation est justifiée quand le temps d'exécution des microroutines est crucial ou si les microroutines sont amenées à travailler très souvent (ex : routines du système).

3.5. Exemple

Le bloc de micro-opérations de la figure II-18 constitue le point de départ de l'exemple. Théoriquement, les micro-opérations devraient être sous forme de langage intermédiaires, mais pour une compréhension plus simple et plus rapide, la forme mnémonique est conservée.

Le graphe de dépendance étant établi à partir de l'utilisation des ressources, les micro-opérations sont balayées séquentiellement et les tableaux (TABMODIF, TABUTIL) sont complétés. Ceux-ci permettent de répondre aux questions de la figure II-14 :

- la ressource est-elle utilisée auparavant ? (UTIL)
- la ressource est-elle modifiée auparavant ? (MODIF).

Pour toute ressource utilisée en des sens différents (en entrée, en sortie) par des micro-opérations différentes, une dépendance existe et le graphe est complété. Ainsi le registre 4 à sa valeur modifiée par la micro-opération 2 et est utilisé par la troisième. La troisième micro-opération dépend donc de la seconde. Le même raisonnement est appliqué au reste du bloc. Le poids des arcs entre micro-opérations vaut 1 lorsque la dépendance est réelle et 0 lorsque les 2 micro-opérations doivent appartenir à la même micro-opération pour assurer une exécution correcte. Ainsi les instructions 18 et 19, qui sont des opérations conditionnelles doivent appartenir à la même micro-instruction que le test (micro-opération 17).

```
(1)     AND (R1,X'OOFF,R14)
(2)     AND(R2,X'OOFF,R4)
(3)     ADD(R4,R14,R14)
(4)     INC(P,P)
(5)     IF(P)
(6)     WAIT
(7)     TRN(MIR,R1)
(8)     TRN(IBR,IR)
(9)     OF(ALU)
(10)    TRN(R14,  )
(11)    WAIT
(12)    TRN(MIR,R5)
(13)    ADD(R5,R14,R14)
(14)    SAMPLE(OVFL)
(15)    OR(R14,X'FFOO,R14)
(16)    INC(P,P)
(17)    TEST(OVFL)
(18)    OF(P,TEST)
(19)    GOTO (  ,  )
```

RESSOURCES	TABUTIF	TABMODIF
R1	1	7
R2	2	
R4	3	2
R5	13	9 12
R14	3 101315	1 3 1315
P	4 5 1618	4 16
IBR	8	5
IR		8
MIR	7 12	5 9 18
OVFT	17	14
MEM	5 9 18	6 11
ALU	9 14	1 2 3 7 10121315

DEGRAPH

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1			1				1												
2			1																
3										1			1						
4					1											1			
5						1	1	1								1			
6									1										
7									1		1								
8																			
9										1	1		1						
10													1						
11																		1	
12													1					1	
13														0	1				
14															1		1		
15																			
16																		1	
17																		0	0
18																			
19																			

ou le "blanc" exprime la non dépendance.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19			
TABTOT	1	1	2	1	2	3	3	3	4	5	5	5	6	6	7	3	7	7	7			
TABTARD	2	3	4	1	2	3	3	7	4	5	5	5	6	6	7	6	7	7	7			
TABCRIT	4	0	5	0	6	7	0	9	0	10	11	12	0	13	14	0	15	17	18	19	0	
DACRIT	4	0	5	0	6	7	0	9	0	10	11	12	0	13	14	0	15	17	18	19	0	

Figure II-18 : exemple de la détection des micro-instructions critiques.

A partir du graphe de dépendance, les micro-opérations critiques (TABCRIT) peuvent facilement être déterminées par l'intermédiaire de TABTOT et TABTARD. Ces micro-opérations critiques doivent encore être analysées pour tenir compte des contraintes hardware. DACRIT est donc le premier schéma du programme. Insérons-y les micro-opérations libres (n'appartenant pas à DACRIT) en utilisant la figure II-17. Au départ, les listes ACTUEL et FUTUR contiennent: ACTUEL: 1 2 4

FUTUR : 3 5

et DACRIT :

1	2	3	4	5	6	7	cycles
4	5	6	9	10 12	13	15	
		7		11	14	17	micro-opérations
						18	
						19	

La première micro-opération est confrontée à la 4ème. Aucun conflit n'apparaissant, elle est insérée dans le cycle 1 et retirée de ACTUEL. La seconde micro-opération est prise en charge. Elle n'appartient pas au chemin critique, mais est en conflit avec la première. Elle est donc placée dans la liste FUTUR. ACTUEL devenu vide, FUTUR y est transférée et la procédure recommence. Finalement DACRIT1 contient la liste des micro-opérations par micro-instruction (= cycle)

DACRIT1 :

cycles	liste des micro-opérations			
1	1	4		
2	2	5		
3	6	7	8	16
4	3	9		
5	10	11		
5'	12			
6	13	14		
7	15	17	18	19

3.6. Algorithme d'optimalisation assurant l'optimum

Dans cet algorithme, on obtient le chemin critique en considérant que des micro-opérations doivent rester dans des cycles donnés (= restrictions). Avant de commencer, la variable NM' est posée égale à l'infini (NM' = 00).

Etape 1 : Obtenir le chemin critique en considérant les restrictions s'il en existe.(voir 2.2).

Etape 2 : Examiner les micro-instructions d'une tranche de temps du chemin critique :

2.1 Il n'existe pas de conflits, un arrangement complet du chemin critique est réalisé. Poser CC_k les micro-instructions appartenant au k^{eme} cycle. Poser NMØ = NM = le nombre total de micro-instructions. Aller à l'étape 3 pour arranger les micro-opérations restantes.

2.2. S'il y a des conflits machine entre K (> 2) micro-opérations, il y a au moins K ! façons de les arranger. Sauver toutes les combinaisons possibles des micro-opérations en conflit. Ces combinaisons constitueront des "forçages" de micro-opérations à des micro-cycles quand un nouveau chemin critique devra être déterminé. Sauver aussi les micro-opérations précédemment forcées. Retourner à l'étape 1 en prenant les micro-opérations précédemment forcées et la première combinaison du nouvel ordonnancement pour trouver un nouveau chemin critique .

Etape 3 : Générer une liste CL_i (vide au début), i = 1, 2, ...NMØ pour l'arrangement futur des micro-opérations libres.

Etape 4 : Pour chaque micro-cycle i(i = 1, 2, 3 ... NMØ), la liste P_i contient les micro-opérations libres pouvant s'exécuter dès le moment i.

4.1. Si P_i est une liste vide, aller à l'étape 5.

4.2. Sinon examiner toutes ses micro-opérations pour détecter les conflits avec celles de CC_i d'abord puis celles de CL_i.

4.2.1. S'il n'y a pas de conflit, ajouter la micro-opération à celle de CL_i .Prendre la micro-opération suivante de P_i et répéter l'étape 4.

4.2.2. S'il y avait un conflit avec une quelconque micro-opération de CC_i (micro-opérations critiques), transporter la micro-opération examinée Me plus loin dans les limites admises par les dates au plus tard, et vérifier à nouveau l'existence de conflits. S'il existe encore au moins un conflit avec des micro-opérations critiques CC répéter le point 4.2.2. S'il n'y en a plus, confronter Me avec les micro-opérations du CL_i correspondant posé CL_j.

4.2.2.1. S'il y a encore des conflits aller en 4.2.3.

4.2.2. Sinon, ajouter Me à CL_j.

Si on ne peut plus retarder Me, générer une nouvelle liste CC_{i+1} (vide) et CL_{i+1} dans laquelle Me est placée. NM = NM + 1.

4.2.3. Si le conflit se produit avec une micro-opération de $(CL_i)_n$, essayer de retarder chacune des micro-opérations dans ses limites autorisées par les dates d'exécution au plus tard dans le but d'obtenir à nouveau un arrangement sans devoir ajouter une nouvelle micro-instruction.

4.2.3.1. Si cela réussit, aller à l'étape 4.

4.2.3.2. Sinon, créer les listes CC_{i+1} (vide) et CL_{i+1} dans laquelle une nouvelle micro-opération libre pourra se placer. Poser $(CC_j)_n$ et $(CL_i)_n$, n = 1, 2... représentant ces copies. Aller à l'étape 4.

Etape 5 : Si toutes les micro-opérations ont été arrangées, comparer NM avec NMØ puis avec NM' précédemment obtenus.

5.1. Si NM = NMØ, alors on a atteint la borne inférieure du nombre total de micro-instructions. Arrêter, car une solution est atteinte.

5.2. Si NM > NMØ et NM > NM' ignorer les calculs actuels et aller en 5.3.1.

5.3. Si NM < NM', Poser NM' = NM et sauver CC_i et CL_i (i = 1, 2 ... NM). Vérifier s'il existe un cas sauvé à l'étape 2.

5.3.1. S'il y a un, aller à l'étape 1 avec les nouvelles micro-opérations critiques forcées.

5.3.2. Si toutes les possibilités ont été examinées, la solution optimale est sauvée. Générer toute les micro-instructions à partir des listes CC_i et CL_i. $M_i = CL_i \cup CC_i$ i = 1, 2 ... NM.

Reprenons le dernier exemple du paragraphe 3.4. Le nouvel algorithme permettra de réduire le nombre de micro-instructions de trois à deux.

La première étape de l'algorithme appliquée à ce bloc donne comme chemin le plus long : $CC'_1 = A$

$CC'_2 = D$

Puisqu'il n'y a qu'une micro-opération dans chaque micro-instruction aucun conflit machine ne peut apparaître (étape 2.1). C'est donc le chemin le plus long sans conflit machine, c'est-à-dire : $CC_i = CC'_i$ pour i = 1, 2 et $NM = NM_0 = 2$. A l'étape 3, deux listes CL sont crées. Au premier passage dans l'étape 4, P_1 contient les opérations B, C et E. Après l'examen de l'instruction B, celle-ci est placée dans CL_1. L'analyse de la micro-opération C fait apparaître un conflit avec B. Le retardement de B (étape 4.2.3.) permet de ne pas créer une nouvelle micro-instruction (CL_i), et C est assigné à CL_1. La situation est donc :

$CC_1 = A \qquad CL_1 = C$

$CC_2 = D \qquad CL_2 = B$

L'opération E peut entrer dans CL_1 et FG dans CL_2.
Toutes les micro-opérations sont placées et NM = NM0 = 2. Il n'est donc pas nécessaire de poursuivre. Le nombre total de micro-instructions est égal à la limite inférieure (NM0 = 2). La solution obtenue est donc optimale.

Chapitre 4

L'adressage

4.1. Le problème

L'exécution des instructions dans un ordre différent de leur apparition donne aux programmes une certaine flexibilité et autorise une compression du code. C'est une propriété bien connue et valable pour tous les langages, mais la flexibilité est un problème crucial en microprogrammation. En effet, ce sont les microprogrammes qui ont la charge de tout le contrôle du processeur , ils doivent pouvoir exécuter telle ou telle partie de code suivant l'instruction qui se présente, suivant l'état du processeur ou enfin suivant une condition. Dans le premier cas, l'adresse du microprogramme à exécuter est formée à partir du code instruction par un décodeur. Les autres cas peuvent se résumer à un choix entre deux adresses. La résolution du problème de l'adressage à l'intérieur des microprogrammes est réalisée de deux façons principales. Dans la première, un registre de micro-instruction donne toujours l'adresse de l'instruction à exécuter et est incrémenté au fur et à mesure. Cela permet de diminuer la longueur de la micro-instruction. Un branchement à deux alternatives nécessite l'utilisation d'un micro-mot pour exprimer la seconde adresse, la première étant obtenue par incrémentation du registre de micro-instruction. La figure II-19(a) illustre ce premier mode de travail. Une seconde façon de résoudre le problème de la séquence d'exécution consiste à utiliser un "modifieur" qui compose une adresse à partir d'entrées multiples présentes ou non. Ce circuit est plus ou moins élaboré selon le nombre de sources acceptées et le type d'opération qu'il effectue entre ces sources (concaténation, ou, et, ou calcul hardware). La figure II-19 (b) montre que d'une façon générale, il s'agit de fusionner par une opération logique un registre R (par exemple celui des interrupts) et un masque M provenant de la micro-instruction pour obtenir une adresse. Afin de ne pas nécessiter trop de "bits" dans la micro-instruction, les différentes adresses ont une partie commune (les poids forts, par exemple).

Le problème étudié dans ce chapitre est d'affecter une adresse à chaque micro-instruction de façon à occuper le mieux possible la mémoire de contrôle et de remplir les champs servant à déterminer la micro-instruction suivante. La solution à ce problème est différente suivant le mode d'adressage envisagé. Dans un premier cas, on affecte des blocs d'adresses à des segments de microprogrammes. Cela revient à découper la mémoire en blocs continus de façon à minimiser le nombre de trous. Dans un second cas, les blocs de mémoire à affecter aux segments sont discontinus. Comme expliqué dans l'annexe 1, la VARIAN travaille suivant le second mode.

4.2. Description de l'algorithme

Alors que la composition travaillait au niveau de blocs de micro-instructions, le problème de l'adressage est résolu au niveau du microprogramme ou plus exactement au niveau de la page de mémoire morte (512 micro-instructions). En effet, pour s'approcher le plus possible d'une occupation maximale de la

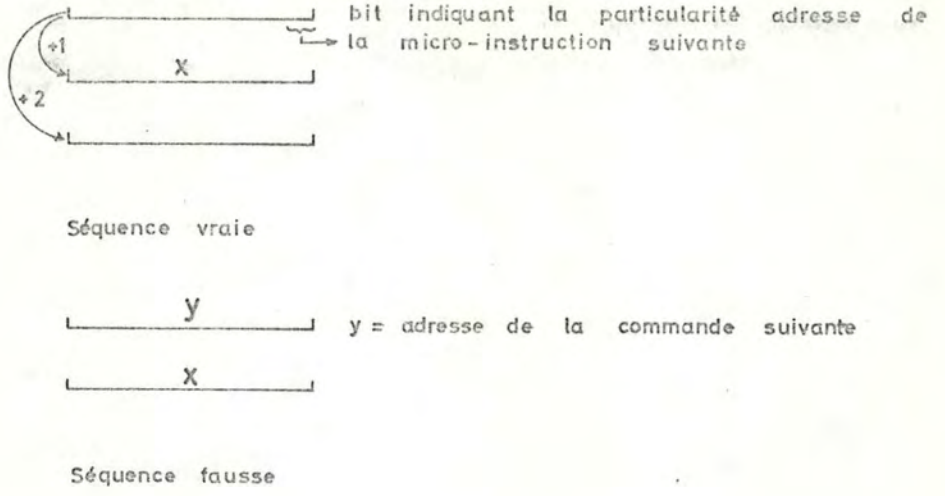
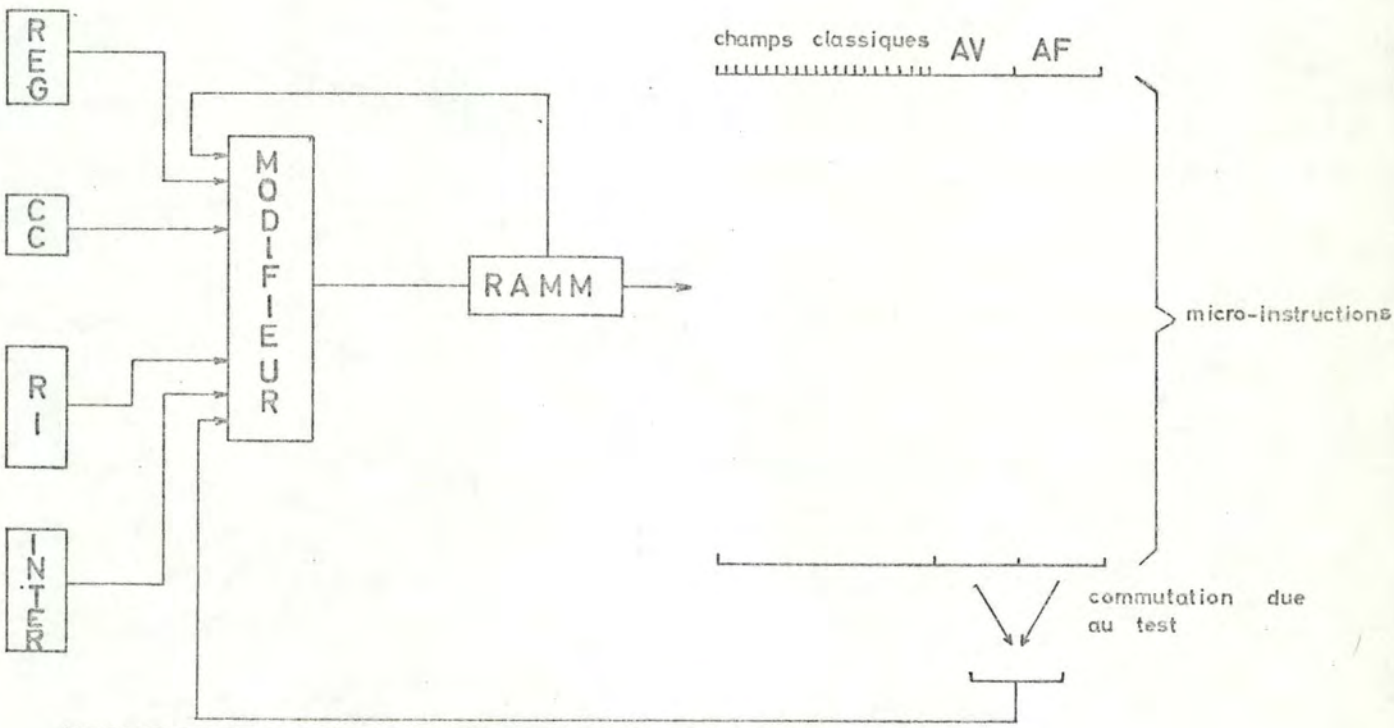


FIG. II - 19 (a) : Adresse interne et disposition verticale



SYMBOLES UTILISES	
REG	Registre
CC	Code condition
RI	Registre instruction
INTER	Registre des interrupts
RAMM	Registre adresse mémoire morte
AV	Adresse de la branche vraie
AF	Adresse de la branche fausse

FIG. II - 19 (b): Adresse interne et disposition horizontale

mémoire de contrôle, il est préférable de laisser le plus possible d'adresses libres plutôt que d'affecter des places, restées libres, au fur et à mesure de la construction des micro-programmes.

Pour assurer la séquence des micro-opérations désirées, il suffit de connaître la succession des micro-instructions. Cela est réalisé grâce à un graphe de séquence. D'un autre côté, l'assignation ne peut s'effectuer aléatoirement, mais au contraire en respectant les contraintes relatives au type d'adressage. Elles s'expriment de deux façons différentes :

- soit en imposant des valeurs à certains bits de l'adresse ;
- soit par la représentation du type d'adressage le plus contraignant effectué sur la micro-instruction.

Partant du langage intermédiaire, du langage source et des tables de symboles et de correspondances (ST et CT), la table des séquences et des contraintes (SG) de la figure II-20 peut être construite (ACTSTO). Une analyse de l'adressage de la VARIAN, montre que tout bit de l'adresse peut :

- être amené à posséder la même valeur que le bit correspondant d'une autre adresse (= pointeur vers adresse de base dans SG) ;
- être forcé à une valeur.

4.2.1. Assignation des adresses (ATSTO)

Tout microprogrammeur a rencontré le problème de l'insertion d'une microroutine dans des places restantes de la mémoire de contrôle. Dans de nombreux cas, il aura dû reprendre en charge toutes les micro-instructions pour analyser les possibilités de permutation et modifier les adresses. Problème d'autant plus épineux qu'il nécessite un nouveau test de toutes les microroutines afin de vérifier le nouvel adressage. Une résolution automatique du problème sera donc appliquée.

Au point de départ, la table de séquences (SG) exprime les contraintes à respecter. Elles sont de deux types :

- des bits déjà forcés à leur valeur définitives ;
- des bits devant posséder la même valeur que les bits correspondants d'une autre adresse.

Avant de commencer l'explication de la procédure suivie, précisons qu'un groupe de micro-instructions est l'ensemble des micro-instructions dont les cinq bits de poids forts doivent posséder la même valeur que ceux d'une autre adresse.

4.2.1.1. Philosophie de la procédure de résolution

L'ensemble des micro-instructions est d'abord divisé en deux :

- les micro-instructions dont les cinq bits de poids forts doivent être identiques (= définition de groupe) ;
- les micro-instructions indépendantes, c'est-à-dire celles n'appartenant pas à un groupe.

Ces deux classes sont traitées séparément. Les micro-instructions indépendantes n'ayant aucune contrainte viendront remplir les

NMI	PREC	SUIV	TY	REF	ADR
(1)		(2)		(1)	

NMI : numéro des micro-instructions

PREC : pointeur vers le premier précédent de la micro-instruction. Le numéro de cette dernière est contenu dans un table où tous les précédents d'une même micro-instruction sont chaînés entre-eux

SUIV : comme PREC pour les suivants

TY : type d'adressage de la micro-instruction

N : adressage normal
C : adressage conditionnel (deux adresses)
S : adressage par sélection de champs (plus de deux adresses)

P_i : changement de page
R : appel de routine

REF : donne le type d'adressage maximal utilisé pour atteindre la micro-instruction. Les types considérés sont :

N : adressage normal, sans contrainte
V : alternative vraie d'un adressage bi-directionnel (GOTO (vrai,faux))
F : alternative fausse d'un adressage bi-directionnel
S : alternative d'un adressage multi-directionnel (GOTO (IRy, MSK ...))
P_i : changement de page
L'ordre de priorité étant N, V/F, S, Pi.

ADR : Partie adresse. Destinée à recevoir les contraintes de positionnement de bits

(1) : Partie utilisée lors de l'assignation des adresses

(2) : Partie utilisée lors de la mise en place des adresses dans les micro-instructions.

Figure II-20 : SG, table des séquences et des contraintes d'adressage.

places restées libres après le traitement des groupes. Les adresses appartenant à des groupes sont, elles, traitées en deux temps. A l'intérieur d'un groupe, les micro-instructions sont d'abord différenciées à l'aide des bits de poids faibles. Plusieurs micro-instructions, de groupes différents ont donc des valeurs identiques dans leurs bits de poids faibles et sont encore susceptibles d'avoir des adresses identiques. La séparation est assurée en affectant des valeurs différentes aux poids forts des différents groupes.

4.2.1.2. Heuristique de résolution

4.2.1.2.1. Affectation d'une valeur aux bits de poids faibles

Cette prodédure travaille au niveau du groupe de micro-instructions. La première tâche évalue le nombre de bits nécessaires pour distinguer les micro-instructions du groupe (solution de : 2^x > nombre de micro-instructions). Il faut alors tenir compte des contraintes. En effet, certains bits peuvent être contraints à posséder la même valeur, certains autres possèdent déjà une valeur. Pour cela, toutes les valeurs qu'il est possible de former à partir des x bits de la micro-instruction sont générées et une combinaison valable est recherchée. Cette génération se fait grâce à deux tables. La première contient toutes les possibilités d'adresses pour les micro-instructions de base des sous-groupes, c'est-à-dire ayant des bits de poids faibles à valeur devant être identique , et toutes les micro-instructions n'ayant que des bits libres ou forcés. L'autre contiendra les micro-instructions reliées à une micro-instruction de base. Elle est organisée en colonnes, chacune d'entre-elle fournissant les adresses des micro-instructions correspondant à un choix dans la micro-instruction de base. La validité de la combinaison est vérifiée par un tableau d'occupation de 2^x positions. Pour diminuer le nombre de combinaisons à générer, les valeurs des micro-instructions n'ayant qu'une possibilité d'adresse sont retirées des tables (= nombre entre parenthèses dans l'exemple).

Exemple : soit le groupe de treize micro-instructions

NMI	ADRESSES								
	8	7	6	5	4	3	2	1	0
1	2	2	2	2		2		2	
2	2	2	2	2		2		2	
3	2	2	2	2		2		2	
4	2	2	2	2		2		2	
5	2	2	2	2		2		2	
6	2	2	2	2		2		2	
7	2	2	2	2		2		2	
8	2	2	2	2	'0'	'0'	'0'	'0'	'0'
9	8	8	8	8	'0'	'0'	'0'	'0'	'0'
10	8	8	8	8	'0'	'0'	'1'	'0'	'0'
11	8	8	8	8	'1'	'0'	'1'	'0'	'0'
12	8	8	8	8	'1'	'0'	'1'	'0'	'1'
.									
.									
.									
17	2	2	2	2					

Les chiffres entre quotes représentant les valeurs imposées aux bits.

Ayant 13 micro-instructions, quatre bits sont nécessaires pour les distinguer. En fait les instructions 10 et 11 ne peuvent pas se distinguer par ces seuls bits. Cinq bits sont donc pris en considération. Deux tables sont alors créées :

table 1

NMI	VAL5
2	2 8 10 (0)
8	0
9	1
10	4
11	20
12	21
17	(0). 1 ... 15

table 2

NMI	$VAL2_i$
1	19 25 27 (17)
3	3 9 11 (1)
4	6 12 14 (4)
5	7 13 15 (5)
6	18 24 26 (16)
7	23 29 31 (21)

NMI : numéros des micro-opérations.
VAL5 : valeurs possibles des 5 bits de poids faibles.
$VAL2_i$: valeurs des 5 bits de poids faibles pour chaque choix d'un VAL5 de la micro-instruction 2.

tableau des combinaisons : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(en hexa)

0	(8) (9) (2) (3) (10) (17) (4) (5)
1	(6) (1) (11) (12) (7)

Si la valeur 2 est prise pour la micro-instruction 2, celle-ci impose les valeurs 3, 6, 7, 18, 19, 23 pour les micro-instructions qui y sont rattachées. Etant encore toutes disponibles, elles leur sont affectées. La micro-instruction suivante (17) peut prendre les valeurs 0 à 15 dans ses 5 bits de poids faibles. Cinq est choisi, les valeurs 0 à 4 étant déjà affectées.
Ce travail effectué pour chaque groupe, il reste à séparer les groupes entre-eux à l'aide des bits de poids forts non encore forcés (4 ou 5).

4.2.1.2.2. Différentiations des groupes

Si leur nombre est inférieur à 16 (2^4) la solution est immédiate puisqu'il existe au moins quatre bits pour les distinguer. Sinon les groupes n'ayant que quatre bits de poids forts non encore forcés sont traités et il faut ensuite espérer qu'il reste assez de valeurs libres pour les affecter aux groupes restants (ayant 5 bits non forcés). Quand il existe plus de 16 groupes dans la première catégorie ou quand il existe plus de 2^5 groupes, certains d'entre eux possèdent la même valeur dans leur cinq bits de poids forts et certaines micro-instructions ont donc la même adresse. Des classes sont alors formées avec les groupes ayant des valeurs identiques dans leur cinq bits de poids forts pour essayer de redistinguer les micro-instructions avec leurs poids faibles (= procédure 4.2.1.2.1.). Cela n'est possible que si les classes ainsi formées ne sont pas trop conséquentes. C'est pourquoi la procédure minimise le nombre de groupes à bits forts identiques.

Cette notion de marche arrière se justifie-t-elle ? pourquoi
En effet, ne pas commencer par distinguer les groupes pour ensuite séparer les adresses à l'intérieur de chacun d'entre-eux ?Pour y répondre, rappelons la philosophie suivie pour créer le module d'adressage :

- le microprogrammeur écrit une nouvelle micro-routine et désire la placer dans la mémoire de contrôle ;
- la mémoire morte est donc complétée module par module et si le problème de l'assignation est simple au départ, il se complique peu à peu.

Il ne faut donc pas utiliser une méthode complexe tant que le problème reste relativement simple, sinon le coût devient prohibitif dans le cas simple. La méthode plus complexe n'est intéressante que là où la précédente échoue.

4.2.1.2.3. Affectation des micro-instructions indépendantes

Le traitement des micro-instructions appartenant à des groupes étant terminés, il reste à affecter une adresse aux micro-instructions indépendantes. Ne possédant aucune contrainte, elles servent à remplir les places laissées vacantes en mémoire morte. Celles-ci sont identifiées dans un tableau d'occupation et le travail est alors très simple.

4.2.2. Mise en place des adresses dans les micro-instructions

(ROTGET2)

Les adresses étant affectées à chaque micro-instruction, le problème est maintenant de les placer dans les micro-instructions afin d'assurer une séquence correcte. La colonne SUIV de la table de séquence fournit l'enchaînement des micro-instructions. Il ne reste plus qu'à fusionner les adresses référencées et à compléter les champs d'adressage convenables. Ceux-ci variant suivant le type d'adressage effectué. La colonne TY du graphe de séquence le définit pour chaque micro-instruction. La figure II-21 fournit la valeur des différents champs suivant le type de l'adressage. Les types C (adressage bi-directionnel) et S (sélection de champs) travaillent sur plusieurs adresses à fusionner. Cette fusion, décrite dans la figure II-21, est directement déduite de la description de l'adressage, particulier au matériel (voir annexe 1).

Le microprogramme représenté par le graphe de la figure II-23 (a) servira de point de départ à un exemple complet de l'adressage. Dans ce graphe, les sommets symbolisent les micro-instructions (MI_i) et les arcs la séquence. Les arcs possèdent également les conditions dans lesquelles ils sont empruntés.

Les colonnes PREC, SUIV, TY et REF de la table des séquences (figure II-23 (b)) se complètent facilement par simple lecture du graphe, seul le remplissage de la colonne ADR présente quelques particularités. L'arc issu de l'entrée et celui issu de MI1 n'ayant aucune condition, les adresses des micro-instructions n'ont aucune contrainte. La MI2 est un branchement sur test, l'alternative fausse doit être paire et par conséquent le bit de la micro-instruction 4 est forcé à zéro. De plus, les 5 bits de poids forts des deux branches de l'alternative doivent être identiques, ce qui s'indique par un pointeur des 5 bits de MI4 vers ceux de MI3. Il en est de même pour les micro-instructions 3 et 4. La micro-instruction 5 est un branchement par sélection de champ de l'IR. Les bits de poids faibles correspondant à des "1" du masque doivent contenir la valeur de la condition du branchement. Comme lors de toute prise en charge d'une micro-instruction, il y a vérification pour détecter les conflits avec ce qui existe déjà. Supposons que le champ TS soit utilisable, les bits correspondant aux zéros du masque doivent tous être identiques, ce qui s'indique par un pointeur vers la branche de valeur nulle (= MI10). Les bits 0 des MI6 et MI7 étant déjà forcés à zéro, les bits 0 des MI8,9 et 10 sont forcés à zéro. Le bit 4 de ces mêmes micro-instructions possèdent déjà un pointeur (vers MI5 et MI11), dans ce cas, le pointeur à placer est transféré dans ces micro-instructions. Il en est de même pour les autres bits de poids forts. Pour traiter MI11, le champ TS est supposé non utilisable et les bits de poids faibles correspondant à des zéros du masque sont positionnés à zéro (bits 0-1-2).

Le graphe balayé, l'assignation d'adresse peut débuter. Le premier groupe, composé de MI1 et MI21 est pris en charge. N'ayant que deux micro-instructions, un seul bit suffit pour les distinguer. Le bit 0 de MI1 étant à 0, celui de MI21 sera positionné à 1. Le deuxième groupe composé de MI2 et MI15 ainsi que de toutes les micro-instructions qui s'y réfèrent directement ou non : MI6, MI7, MI8, MI11, MI5, MI9, MI10 et MI19. Quatre bits

TYPE	Valeurs prises par les champs																			
	TS				AF					MS				MT	FS					
N	0	0	0	0	(8)	(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)	0	0	0	0	0		
C	0	(2)	(1)	(0)	(8)	(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)	0	0	0	0	0		
		(*)									(+)									
P_i	i	i	i	i	(8)	(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)	0	0	0	0	0		
S	4'	4'	4'	0	(8)	(7)	(6)	(5)	2'	5'				3'	5'					

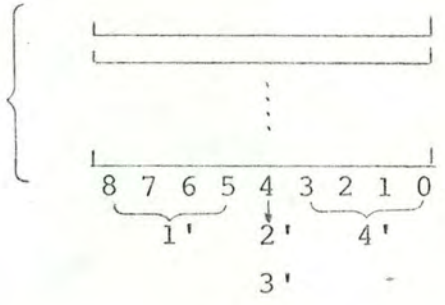
Les nombres entre parenthèses font référence aux bits correspondants de l'adresse

(*) de branchement si la condition n'est pas vérifiée

(+) de branchement si la condition est vérifiée.

Adressage par sélection de champs (S)

ensemble des adresses référencées



1' Les 4 bits de poids forts communs se placent dans AF

2' Si le bit 4 est commun, le placer dans AF

3' Si le bit 4 n'est pas commun, mettre 0 dans le bit 0 de AF et 1 dans MT

4' Si TS est utilisable, mettre les bits correspondants à des 0 dans les bits 3, 2 et 1 du masque. Sinon laisser TS inchangé

5' Masque positionné dans une phase précédente.

Figure II-21 : Insertion des bits de l'adresse dans les champs.

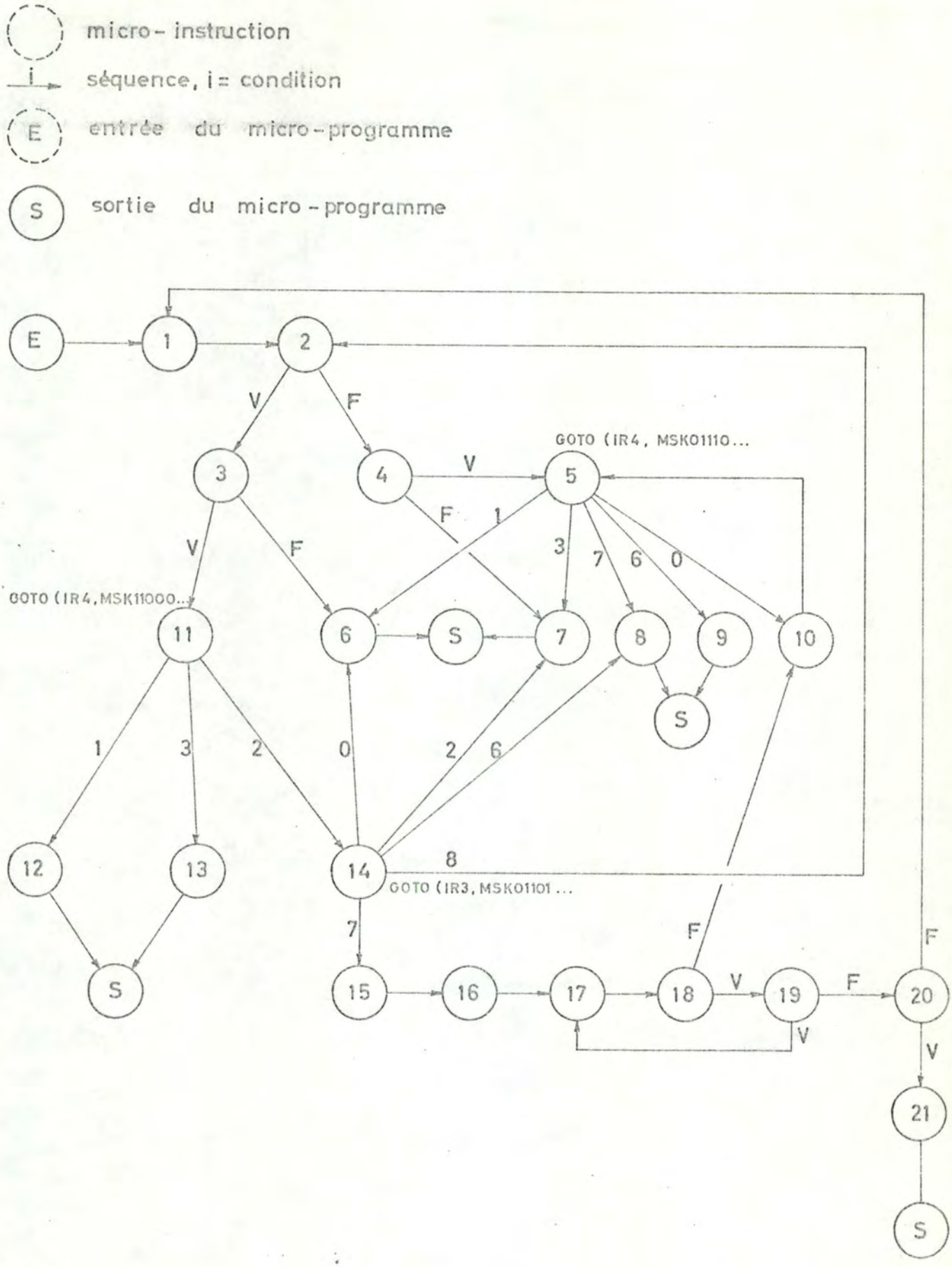


FIG. II - 23 (a) : Graphe du programme

NMI	PREC	SUIV	TY	REF	ADR								
					8	7	6	5	4	3	2	1	0
1	E, 20	2	N	⊬ F	21	21	21	21	21				"0"
2	1, 14	3,4	C	⊬ S	6	6	6	6	6	"1"	"0"	"1"	"0"
3	2	6,11	C	V									
4	2	5,7	C	F	3	3	3	3	3				"0"
5	4,10	6,7,8,9,10	S	V	10	10	10	10	10				
6	3,5,14	S	N	⊬ S	11	11	11	11	11	"0"	"0"	"1"	"0"
7	4,5,14	S	N	⊬ S	5	5	5	5	5	"0"	"1"	"1"	"0"
8	5,14	S	N	S	10	10	10	10	10	"1"	"1"	"1"	"0"
9	5	S	N	S	10	10	10	10	10	"1"	"1"	"0"	"0"
10	5,18	5	N	S	19	19	19	19	19	"0"	"0"	"0"	"0"
11	3	12,13,14	S	V	10	10	10	10	10				
12	11	S	N	S					"0"	"1"	"0"	"0"	"0"
13	11	S	N	S	12	12	12	12	"1"	"1"	"0"	"0"	"0"
14	11	2,6,7,8,15	S	S	12	12	12	12	"1"	"0"	"0"	"0"	"0"
15	14	16	N	S	6	6	6	6	6	"1"	"1"	"1"	"1"
16	15	17	N	N									
17	16,19	18	N	⊬ V									
18	17	10,19	C	N									
19	18	17,20	C	V									
20	19	1 ,21	C	F	17	17	17	17	17				"0"
21	20	S	N	V									

Dans ADR : chiffre : pointeur
"chiffre" = valeur du bit.

Figure II-23 (b) : SG, graphe de séquence.

devraient suffire pour les distinguer par leur adresse. Une table de possibilités et le tableau des combinaisons sont construits.

2	10
5	0, 1, ... 15
6	2
7	6
8	14
9	12
10	0
11	0, 1, ... 15
15	15
19	0, 1, ... 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	(5)		(11)	(19)											
(10)		(6)				(7)				(2)		(9)		(8)	(15)

table 1 : des possibilités tableau d'occupation

Les possibilités uniques sont directement placées dans le tableau d'occupation (10, 2, 6, 12, 14, 0 et 15). Les adresses restées libres peuvent facilement recevoir les adresses des micro-instructions "flottantes". La table 2 des possibilités est ici inexistante, car aucun des bits de poids faibles ne contient de pointeur. Il en est de même pour les autres groupes.

N'ayant que cinq groupes, il n'y aura aucune difficulté pour les séparer avec les 4 bits de poids forts. Il reste alors à placer les micro-instructions indépendantes dans les places restées libres. Toutes ces adresses se retrouvent dans la table des adresses AT (figure II-23 (c)).

Ces adresses doivent maintenant être placées dans les champs du langage intermédiaire (MIT). La colonne SUIV estutilisée pour retrouver la séquence. Dans ce qui suit, une micro-instruction de chaque type est expliquée (TY).

type normal N : (exemple MI1). L'adresse de la micro-instruction suivante (MI2) est placée telle quelle dans les champs AF et MS.

type C (exemple MI2) : Deux adresses doivent être fusionnées : celles de MI3 et MI4. Lors de la recherche de leur adresse, leur type de référence est analysé afin de connaître la destination des bits de poids faibles (MS ou TS). Les cinq bits de poids forts, communs, sont placés dans AF.

type S (exemple MI5). Cette fois, plusieurs adresses doivent être fusionnées. MI6 est d'abord prise en charge, ses 4 bits de poids forts sont placés dans AF. Pour ce groupe, TS a été suppposé utilisable, il faut donc placer les bits correspondants à des zéros du masque (= champs MT-MS) dans TS. Les autres micro-instructions (MI7, MI8, MI9, MI10) sont alors analysées pour détecter une différence éventuelle dans la valeur de leur bit 4. Celle-ci n'apparaissant pas dans ce cas, il est placé dans le champs AF.

	8	7	6	5	4	3	2	1	0		
1	0	0	0	0	0	0	0	0	0	00	1
2	0	0	0	1	0	1	0	1	0	2A	2
3	0	0	1	0	0	0	0	0	1	41	3
4	0	0	1	0	0	0	0	0	0	40	4
5	0	0	0	1	0	0	0	0	1	21	5
6	0	0	0	1	0	0	0	1	0	22	6
7	0	0	0	1	0	0	1	1	0	26	7
8	0	0	0	1	0	1	1	1	0	2E	8
9	0	0	0	1	0	1	1	0	0	2C	9
10	0	0	0	1	0	0	0	0	0	20	10
11	0	0	0	1	0	0	0	1	1	23	11
12	0	0	1	1	0	1	0	0	0	68	12
13	0	0	1	1	1	1	0	0	0	78	13
14	0	0	1	1	1	0	0	0	0	70	14
15	0	0	0	1	0	1	1	1	1	2F	15
16	0	0	0	0	0	0	0	1	0	O2	16
17	0	1	0	0	0	0	0	0	1	81	17
18	0	0	0	0	0	0	0	1	1	O3	18
19	0	0	0	1	0	0	0	0	0	24	19
20	0	1	0	0	0	0	0	0	0	80	20
21	0	0	0	0	0	0	0	0	1	01	21
S	0	0	0	0	0	0	1	0	0	04	S

Figure II-23 (c) : AT, table des adresses

TS	AF	MS	MT	FS
	00010	1010		
0000	00100	0001		
0001	00010	0011		
0011	00010	0001		
0 0	00010			
	00000	0100		
	00000	0100		
	00000	0100		
	00000	0100		
	00010	0001		
	00110		1	
	00000	0100		
	00000	0100		
0001	00010			
	00000	0010		
	01000	0001		
	00000	0011		
0000	00010	0100		
0000	01000	0001		
0000	00000	0001		
0000	00000	0100		

complété lors de l'analyse sémantique

Figure II-23 (d) : MIT, langage intermédiaire (extrait)

T R O I S I E M E P A R T I E

Champ d'application de notre langage et conclusions

Chapitre 1 La microprogrammation dans le cadre d'un compilateur de langage de haut niveau

1.1. Enoncé du problème

1.2. Etude comparée des différentes solutions
1.2.1. Les différentes solutions
1.2.1.1. La méthode classique : quadruples - langage de base - microprogrammes
1.2.1.2. Interprétation des quadruples par microprogrammes
1.2.1.3. Traduction directe des quadruples en micro-instructions
1.2.2. Analyse comparée des temps d'exécution

1.3. Choix de la meilleure solution

Chapitre 2 Conclusions et réalisations

Chapitre 1

La compilation dans le cadre d'un compilateur de langage de haut niveau

1.1. Enoncé du problème

Un modèle idéal de compilateur d'un langage de microprogrammation de haut niveau a été établi dans la première partie (figure I-6). Rappelons qu'il génère des micro-instructions en trois grandes étapes :

- la première travaille sur un langage source plus ou moins complexe - c'est-à-dire libérant l'utilisateur des contraintes inhérentes au matériel utilisé - et fournit une séquence de micro-opérations équivalente ;
- la seconde reprend l'ensemble des micro-opérations pour les fusionner, en tenant compte des concurrences dans l'utilisation des différentes ressources.
- finalement les contraintes du matériel sont prises en compte pour créer un microprogramme exécutable par la machine envisagée.

Dans la deuxième partie du mémoire, nous avons construit les algorithmes nécessaires pour transformer les séquences de micro-opérations en micro-instruction. Nous avons ainsi mis en évidence comment réaliser les deux dernières étapes du modèle général. Avant de conclure, il reste encore à montrer que les micro-opérations imaginées sont effectivement générables par programme. Ce programme peut tout aussi bien être la première étape du modèle idéal établi (figure I-6) que la phase de traduction d'un langage intermédiaire (quadruples, triples, polonais inversé ...) en langage de base dans un compilateur classique. C'est ce dernier qui est envisagé et développé ci-dessous. Nous en profitons pour mettre en évidence les gains obtenus par l'introduction de la microprogrammation dans les compilateurs.

Rappelons qu'un compilateur est un système dont la fonction est de traduire un programme source exprimé en langage de haut niveau (FORTRAN, BASIC, COBOL, ...) en un programme objet reconnaissable par une configuration spécifique. La traduction s'effectue généralement en trois phases. La première est une analyse syntaxique des instructions du langage source assurant leur validité et établissant les tables de symboles ; la deuxième phase consiste en une analyse sémantique qui convertit le langage source en un langage intermédiaire. Les quadruples constituent une représentation classique des instructions binaires du texte intermédiaire. La troisième phase, appelée processus de génération de code, convertit le langage intermédiaire en un code machine particulier. Le processus de conversion est donc fortement lié à l'ensemble des instructions de la machine pour laquelle il est écrit.

Trois modèles seront envisagés. Le point de départ est toujours l'ensemble des quadruples représentant le programme. On se rappelera qu'un quadruple est composé d'un code opération et de un à trois opérandes. La structure de base est :

"(<code opération>, <opérande>, <opérande>, <résultat>)" chacun des opérandes étant spécifiés par son nom ou par un pointeur. Les opérations envisagées sont reprises et expliquées à la figure III-1. La représentation physique des quadruples y est également reprise.

Opérations	Opérandes	Actions
BR	i, ,	Brancher au quadruple i
BRL	P, ,	Brancher au quadruple décrit par l'élément pointé par P
BZ BP BN	i,P,	Brancher au quadruple i si la valeur de l'élément pointé par P est nulle, positive ou négative
BG	i,P1,P2	Brancher au quadruple i si la valeur pointée par P1 est supérieure, égale ou inférieure à celle pointée par P2
+/-	P1,P2,P3	(P1) +/- (P2) -----→ (P3)
*/÷	P1,P2,P3	(P1) */÷ (P2) -----→ (P3)
:=	P1,P2,	Affecter la valeur pointée par P1 à l'élément pointé par P2

code opération	1er opérande (=adr-1)	2ème opérande (=adr-2)	3ème opérande (=adr-3)
(2 bytes)	(2 bytes)	(2 bytes)	(2 bytes)

Code opérations :

bits : 15, 14, 13 } signalent les adresses existantes

12 vaut 1 si adr-1 est un numéro de quadruple

11, 10 } classe du quadruple

9, 8 } identification du quadruple

7 ↓ 0 } non utilisée

Classe et identification du quadruple

BITS					OPERATIONS
12	11	10	9	8	
1	0	1	0	0	BR,i, ,
0	0	1	0	0	BRL,P, ,
1	0	1	0	1	BZ,i,P,
1	0	1	1	0	BP,i,P,
1	0	1	1	1	BN,i,P,
0	0	0	0	0	+,P1,P2,P3
0	0	0	0	1	-,P1,P2,P3
0	0	0	1	0	*,P1,P2,P3
0	0	0	1	1	÷,P1,P2,P3
0	1	0	-	-	:=,P1,P2,
1	1	1	0	0	BG,i,P1,P2
1	1	1	0	1	BL,i,P1,P2
1	1	1	1	0	BE,i,P1,P2

Figure III-1 : Description des quadruples.

Certaines méthodes partent du langage source (WEBER,1967 ; BASHKOW, 1967), mais elles semblent peu intéressantes par suite de la dépendance des microprogrammes vis-à-vis du langage machine. Nous ne nous préoccupons donc pas des phases syntaxique et sémantique qui sont laissées aux compilateurs existants.

Pour apprécier l'efficience de ces différents modèles, nous avons, pour chacun d'eux, établi les temps d'exécution des divers quadruples.

1.2. Etude comparée des différentes solutions

1.2.1. Présentation des différentes solutions

1.2.1.1. La méthode classique : quadruples - langage de base - microprogrammes

La plupart des compilateurs actuels (GRIES,1971) après avoir généré des quadruples, les traduisent en un langage de base spécifique et c'est ce langage qui finalement permet d'accéder aux microprogrammes. Le modèle représentatif de cette solution a été construit suivant la méthodologie de Gries en profitant de la remanence de l'accumulateur et en utilisant des routines d'optimalisation du nombre de variables temporaires.

La traduction ne pose pas de problèmes ; la génération se conformant à l'ordinogramme de la figure III-2 pour un langage à un opérande comme celui de la VARIAN. En effet, selon l'opérateur du quadruple, on distingue trois familles :

les quadruples - d'assignation (:=) ;
- de branchements ;
- des opérations arithmétiques.

-L'assignation, définie par le quadruple (:=, P1, P2,) devra garnir la zone "valeur" de l'identificateur pointé par P2 dans la table des symboles avec la valeur de la table des constantes pointée par P1. Il s'agit donc de convertir le quadruple en un
LDA symb + P1
STA const + P2.

-Les branchements conditionnels et inconditionnels. Pour un branchement inconditionnel, la valeur pointée par le deuxième opérande du quadruple constitue le déplacement ou une adresse relative.
Dans le cas d'un branchement conditionnel, on génère :

```
LDA     P1
SUB     P2
JA  { P                  si l'accumulateur est libre
    { N     i
    { Z
```

-Les opérations arithmétiques
La traduction en langage machine ne pose pas de problème si ce n'est celui de l'optimalisation si la nature du problème l'exige. Ceci serait le cas pour un programme du type :

X = (D+E) + (A * B)-C

Y = (A*B)/(D+E)/F) * C

Z = A * B * C

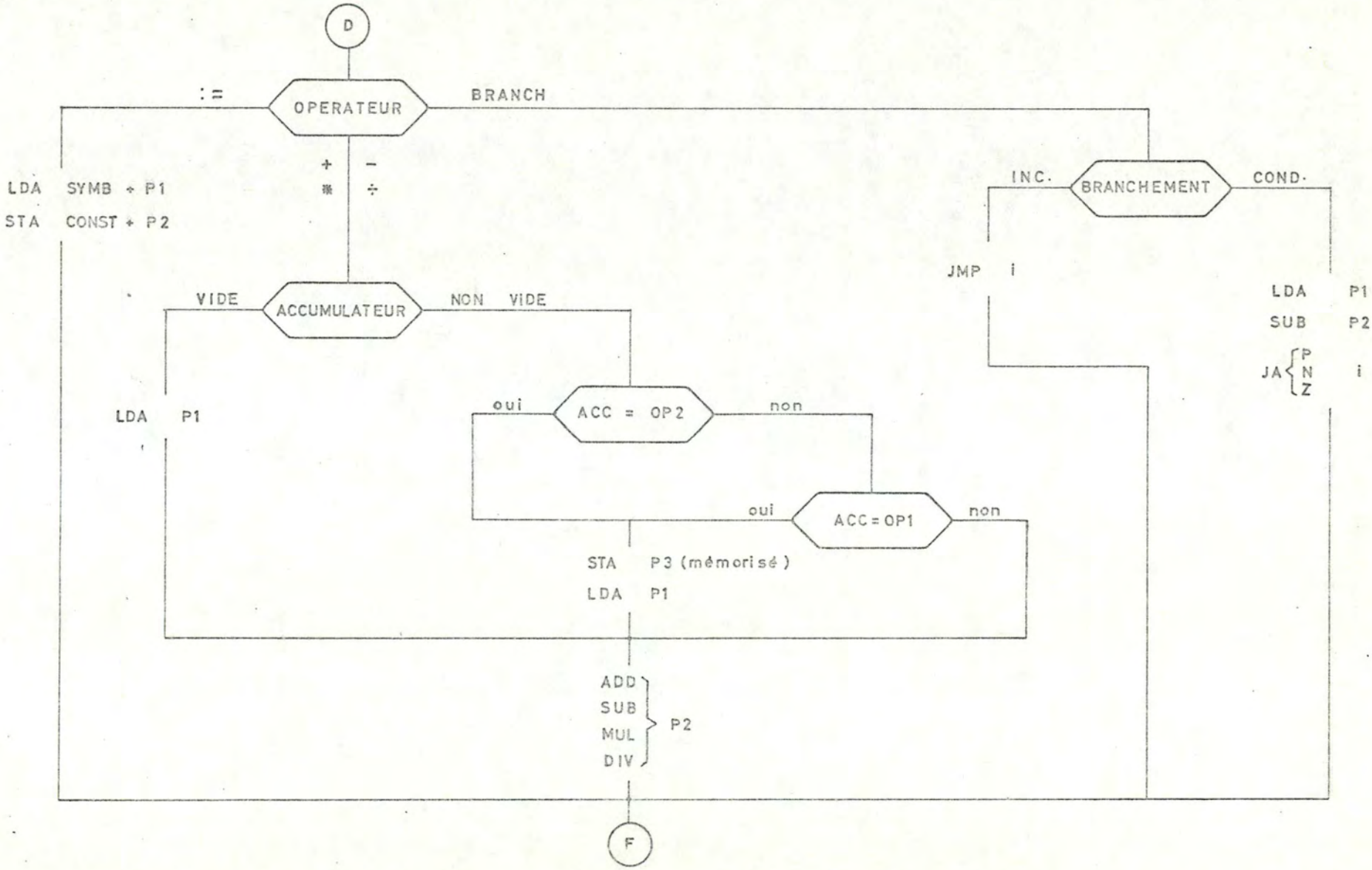


FIG. III - 2 : Génération des instructions VARIAN à partir des quadruples

L'optimalisation consistera à économiser du volume, d'une part en réduisant le nombre des instructions (suppression des instructions STA et LDA qui conduisent à des opérations blanches) et d'autre par en diminuant le plus possible le nombre de zones de stockage des variables intermédiaires (GRIES, 1971).

Les temps d'exécution résultants sont donnés à la figure III-3. Pour les établir, le langage de base de la VARIAN a été utilisé ce qui permet de reprendre les temps donnés par le constructeur pour évaluer le nombre de manosecondes nécessaires à l'exécution des instructions remplaçant les différents types de quadruples.

1.2.1.2. Interprétation des quadruples par microprogrammes

L'exécution microprogrammée demande un accès mémoire pour obtenir le texte intermédiaire, décoder le code opération et brancher au microprogramme approprié pour transformer les données ou modifier la séquence. C'est exactement la procédure qu'un ordinateur suit pour exécuter ses instructions. Si la machine possède un ensemble de registres beaucoup plus rapidement accessibles que la mémoire, des routines spéciales sont nécessaires pour accéder à ces registres et traiter les cas où les données s'y trouvent. La disposition de registres généraux complique aussi la seconde phase de l'analyse sémantique de la compilation. En effet, le texte intermédiaire doit être généré en tenant compte de l'utilisation des registres pour sauver les résultats intermédiaires. Une approche simple à ce problème est de permettre au compilateur de générer les instructions du texte intermédiaire sans tenir compte de ce fait et employer une seconde phase qui essaye de mettre les résultats intermédiaires dans les registres libres. Cette technique n'est pas développée ici.

L'approche de l'exécution directe est fortement influencée par le matériel utilisé. Une analyse détaillée a été faite pour la séquence des opérations nécessaires à l'exécution d'une série de quadruples. Ceux-ci sont supposés placés en mémoire. Une routine de "fetch" prend un quadruple de la mémoire centrale et positionne un pointeur vers le quadruple suivant. Une routine de décodage et de sélection balaye la partie opération et branche vers une séquence de micro-instructions qui exécute l'opération demandée ou initialise un branchement c'est-à-dire une opération mémoire. Toute exécution de routine se termine par un branchement vers la routine de "fetch". L'exécution d'un quadruple peut donc être décomposée en quatre cycles principaux :

- prise en charge du code opération ;
- prise en charge des opérandes ;
- exécution ;
- sauvetage des résultats.

Afin de déterminer les temps d'exécution des différents quadruples, les microroutines nécessaires ont été écrites. Le nombre de micro-instructions servant à les exécuter a été déterminé et multiplié par le temps de base de la mémoire de contrôle (165 manosecondes). Les résultats sont présentés à la figure III-4.

QUADRUPLES	Génération		temps	
	mini	maxi	mini	maxi
:=,P1,P2	sta	lda sta	1320	2640
BR,i, ,	jmp	jmp	1320	1320
BG,i,P1,P2 BL,i,P1,P2 BE,i,P1,P2	sub ja	lda sub ja	2640	4620
BP,i,P, BN,i,P, BZ,i,P,	ja	lda ja	1320	3300
+,P1,P2,P3	add sta	lda add sta	2640	3960
-,P1,P2,P3	sub sta	lda sub sta	2640	3960
* ,P1,P2,P3	lda mul sta	lda stb ldb mul sta	7466	10601
÷,P1,P2,P3	lda div sta	lda stb ldb div sta	7796	11261

Figure III-3 : Temps d'exécution (mano-s) des quadruples par la méthode classique.

Quadruples	temps
:=,P1,P2	2640
BR,i, ,	1650
BG,i,P1,P2 BL,i,P1,P2 BE,i,P1,P2	3960
BP,i,P BN,i,P BZ,i,P	2970
+,P1,P2,P3	3795
-,P1,P2,P3	3795
* P1,P2,P3	8745
$\frac{:}{.}$,P1,P2,P3	9240

Figure III-4 : temps d'exécution des quadruples interprétés par microprogramme.

Quadruples	Nbr. de micro-instr.		temps	
	mini	maxi	mini	maxi
:=,P1,P2	1	8	165	1320
BR,i, ,	1	1	165	165
BR,i,P1,P2 BL,i,P1,P2 BE,i,P1,P2	2	12	330	1980
BP,i,P, BN,i,P, BZ,i,P,	2	7	330	1155
+,P1,P2,P3	1	14	165	2310
-,P1,P2,P3	1	14	165	2310
*,P1,P2,P3	31	50	5115	8250
$\frac{:}{.}$,P1,P2,P3	34	61	5610	10065

Figure III-5 : Temps d'exécution (mano-s) des quadruples traduits directement en micro-instructions.

1.2.1.3. Traduction directe des quadruples en micro-instructions

La philosophie de la traduction est identique à celle utilisée pour traduire les quadruples en langage de base. La micro-instruction, ou plutôt la micro-opération, est directement utilisée comme langage de base. Une différence fondamentale existe cependant : le travail au niveau de la mémoire de contrôle permet, dans bien des cas, l'utilisation de beaucoup plus de régistres qu'au niveau du langage de base. Une gestion de ces régistres, dans le but de minimiser les accès à la mémoire centrale, est donc intéressante.

Cette routine ainsi que celles nécessaires à la génération ont été écrites afin de déterminer les micro-opérations nécessaires. Ces dernières furent ensuite regroupées en micro-instructions pour déterminer les temps minimaux (les opérandes se trouvent dans les registres) et maximaux (pas d'opérandes dans les registres) d'exécution de chaque quadruple (figure III-5).

1.2.2. Analyse comparée des temps d'exécution

Les temps d'exécution des quadruples demandés par les trois modèles ont été rassemblés dans le tableau récapitulatif qui suit. On remarque que ces temps donnés dans la première et la dernière colonne de ce tableau sont des temps moyens arrondis à un nombre entier de micro-instructions.

Tableau récapitulatif des temps d'exécution (nanosecondes)

QUADRUPLES	METHODE CLASSIQUE	INTERPRETATION PAR MICROPROGRAMME	GENERATION DES MICRO-INSTRUCTIONS
:=,P1,P2,	1980	2640	825
BR,i, ,	1320	1650	165
BG i, P1,P2	3630	3960	1155
BP i, P1,	2310	2970	825
-/+,P1,P2,P3	3300	3795	1320
*, P1,P2,P3	9075	8745	6765
÷ , P1,P2,P3	9570	9240	7920

Comparant ces résultats on constate ce qui suit :

- Modèle 1 et 2 : Excepté les multiplications et les divisions, tous les autres quadruples ont un temps d'exécution supérieur dans le cas d'une interprétation directe par microprogramme. Cela provient des "fetch" d'opérandes puisque l'on travaille en fait sur des instructions à trois adresses tandis que le premier modèle travaille sur des instructions à une adresse. Les temps donnés pour le deuxième modèle pourraient cependant être diminués par une utilisation et une gestion de tous les registres ce qui est toujours impossible dans le premier cas. En conclusion, nous pouvons dire qu'au point de vue temps d'exécution, les deux modèles pourraient être identiques. Dans les machines commercialisées, le premier modèle sera toujours le meilleur, mais

pour le concepteur de machines COBOL, FORTRAN ou autres, l'interprétation directe peut être intéressante du point de vue place occupée dans la mémoire de contrôle. Broca et Merwin (BROCA,1973) ont démontré que cette solution pouvait être intéressante dans certaines conditions.

+ Modèle 1 et 3 : le gain obtenu par la troisième solution est évident. Il est principalement dû à l'utilisation de tous les registres disponibles. De plus, il faut remarquer que les temps donnés pour le troisième modèle sont surestimés, car lors de leur établissement les fusions possibles entre groupes de micro-opérations n'ont pas été considérées. L'inconvénient majeur du troisième modèle est le grande taille des séquences générées. Dans bien des cas il faudrait envisager un chargement dynamique de la mémoire de contrôle, ce qui peut diminuer considérablement les gains obtenus.

1.3. Choix de la meilleure solution

On a vu qu'à l'aide du troisième modèle il est possible de générer des séquences de micro-instructions. Pour des programmes de grande taille, il se peut que la traduction du langage de haut niveau en micro-instructions ne soit pas intéressante ; toutefois elle le sera certainement dans le cas de programmes plus courts. Dans ses travaux, Jacquemart (JACQUEMART, 1975) isole des séquences de quadruples se retrouvant le plus souvent dans les différentes applications d'un Centre de traitement de l'information. Ces quadruples seront microprogrammés dans l'intention de diminuer les temps d'exécution. Ces séquences n'étant pas trop longues, l'intérêt est évident.

Nous pouvons donc dire qu'à partir des idées de Jacquemart et à l'aide du troisième modèle, l'intégration du hardware et des différents niveaux du software en un système global serait facilement réalisable en se servant du compilateur décrit dans la seconde partie.

La génération directe des micro-instructions (sans passer par les micro-opérations) est d'une part beaucoup plus difficile à réaliser et d'autre part peu efficace. En effet, une génération directe ne permet plus d'effectuer de regroupements comme c'est le cas en travaillant à partir des micro-opérations. De plus, la plus grande part de la flexibilité vis-à-vis des contraintes du matériel est perdue. Le langage créé est donc un point de passage facile et permet une certaine indépendance vis-à-vis du hardware.

Chapitre 2

Conclusions et réalisations

Actuellement le terme "microprogrammation" évoque l'image d'un travail très ardu demandant une connaissance détaillée des particularités du hardware et utilisant des moyens relativement primitifs. L'acceptation généralisée de cet outil par le programmeur requiert la création de supports plus sophistiqués. Traditionnellement, l'aide à la programmation est fournie par un système de traduction d'un langage de haut niveau. Celui-ci permet une meilleure concentration de l'attention sur la logique du travail et non plus sur les caractéristiques du hardware. Ces avantages devraient être appliqués à la microprogrammation. C'est l'idée qui nous a conduit durant la réalisation de ce mémoire.

Nous avons commencé par effectuer un tour d'horizon des langages de microprogrammation existants. Il est rapidement apparu qu'ils sont encore de bas niveau, c'est-à-dire que l'utilisateur doit connaître le matériel sur lequel il travaille et tenir compte de toutes les contraintes. Cet examen a permis :

- de mettre en évidence les principaux problèmes qui se présentent ;
- d'établir les objectifs d'un modèle idéal.

Ceux-ci peuvent se résumer à la recherche d'un compromis entre :

1. Une dépendance de la machine ;
2. La facilité de détection et de la représentation du parallélisme ;
3. Le naturel de l'expression.

L'indépendance peut être obtenue en adaptant les principes de la théorie des compilateurs à phases multiples, c'est-à-dire en travaillant sur une succession de langages intermédiaires. Ce modèle comporte alors trois parties principales .

1. L'analyse syntaxique et sémantique ;
2. La composition de micro-instructions à partir des micro-opérations ;
3. La création des micro-instructions conformément aux contraintes imposées par le hardware.

La première phase comprend de nombreux problèmes de gestion des ressources offertes à l'utilisateur, mais inexistantes au niveau des micro-opérations définies. Ainsi, par exemple, si le nombre de registres alloués au microprogrammeur est supérieur à 16 (=nombre de registres hardware), il faut pouvoir traduire le travail défini en une séquence effectuant ce même travail, mais cette fois en utilisant uniquement les 16 registres. Nous avons simplement montré qu'il était possible de traduire directement un langage symbolique en micro-opérations (cf. chapitre III-1). Par contre, nous avons élaboré les deux dernières parties dans le détail en appliquant l'étude à une machine existante : la VARIAN 73.

Dans notre système, il n'appartient pas à l'utilisateur d'exprimer le parallélisme qui conduirait à une exécution optimale. Cet objectif est atteint par les micro-opérations qui, dans la phase de composition, sont regroupées en utilisant toutes les possibilités de la machine et prenant en compte certaines contrain-

tes. Cette phase de composition établit un graphe de dépendance des micro-opérations et, après une recherche des micro-instructions critiques, construit un premier schéma du microprogramme sans tenir compte encore des contraintes du matériel. La prise en charge des contraintes propres à la machine conduit à la construction d'un nouveau schéma qui supprime les conflits entre opérations critiques et insère les micro-opérations libres. Signalons toutefois que toutes les contraintes existantes n'ont pas été prises en charge. Ainsi, l'utilisateur doit toujours connaître un sous-schéma du chemin des données et c'est à lui d'assurer la synchronisation lorsqu'il lance une opération mémoire.

Les microprogrammes ainsi construits ne sont pas toujours optimum, c'est-à-dire que le nombre de micro-instructions générées peut être supérieur au minimum nécessaire, mais nous avons montré (cf. II-3.6) ce que la recherche d'un optimum absolu pouvait coûter en temps de compilation et en occupation mémoire.

Avant de construire les micro-instructions définitives, il reste à assigner une adresse à chacune d'entre-elles. Ce problème, n'est pas simple car il faut essayer d'atteindre un bon rapport d'utilisation de la mémoire de contrôle. Nous partons d'un tableau représentant les adresses des différentes micro-instructions. Dans un premier temps, les contraintes d'adressage sont traduites en forçant la valeur de bits ou en obligeant certains d'entre-eux à posséder la même valeur que d'autres (= bits liés). Des ensembles, appelés groupes, de micro-instructions devant posséder des bits à valeur identique sont alors mis en évidence et les adresses de leurs micro-instructions sont différenciées par les bits restés libres. Il suffit à ce moment de distinguer les groupes pour être sûr que toutes ces micro-instructions ont des adresses différentes. Cela s'effectue en agissant sur les bits liés. Les micro-instructions n'ayant aucune contrainte peuvent alors recevoir l'adresse d'une des positions mémoire restées libres.

Le modèle décrit n'a pas été réalisé dans son entièreté. On remarquera cependant que le langage (intermédiaire) qui nous sert de point de départ est d'un niveau assez élevé (cf. fig. I-3) puisque certaines micro-opérations sont de véritables "macros". De plus il est présenté sous une forme mnémonique très semblable à celles des langages de base. En outre, on n'exige plus de l'utilisateur qu'une connaissance partielle du hardware.

Tous les principes de la traduction de ce langage en micro-instructions ont été décrits dans le mémoire. Afin de ne pas en alourdir la lecture, les détails complémentaires ainsi que les programmes relatifs à la composition des micro-opérations en micro-instructions sont reportés dans des annexes séparées. Celles-ci sont disponibles au secrétariat.

B I B L I O G R A P H I E

AHO, A.V., SETHI, R. et ULLMAN, J.D."A formal approach to code optimization"
SIGPLAN Proceedings of a symposium on Compiler Optimization, juillet 1970.

ALLEN, F.E. "Control Flow analysis"
ACM SIGPLAN Notices, Vol.5 N7, july, 1970.

AMDAHL, L.D. "Microprogramming and stored logic"
Datamation, vol 10 N2, Février, 1964.

ATSTOPAS, F.F. and BLUKAS, K.I. "Method of minimizing computer microprograms"
Automatic control, Vol. 5 N° 4, pp. 10-16, 1971.

BARTOW, N. et Mc GUIRE, R. "System 360/85 microdiagnostics".
Spring joint computer conférence, AFIPS Proceedings, Vol. 36, pp. 191-197, 1968.

BASHKOW, T., SASSON, A. et KRONFELD, A. "System design of a FORTRAN machine"
IEEE transaction on computers, août 1967.

BECK, L. et KEELER, P. "The C-8401 data processor"
Datamation,Vol. 10 N° 2, février 1964.

BERNSTEIN, A.J. "Analysis of program for parallel processing"
IEEE trans.Electronics computer.
Vol. EC-15, PP. 757-763, octobre 1966.

BOUTWELL, J.R. "The PB 440 computer"
Datamation, Vol. 10 N° 2, février 1964.

BROCA, F.R. et MERWIN, R.E. "Direct microprogrammed execution of the intermediate text from a high-level language compiler"
Proceedings of ACM SIGPLAN-SIGMICRO interface meetings, pp. 145-153, juin 1973.

CHAMBERLIN, D.D. "The single assignment approach to parallel processing"
Proceedings of the fall joint computer conferencœ, Vol. 39, 1971.

CHU, Y. "Recursive microprogramming in a syntax recognizer".
MICRO-5 Preprints, pp. 91-98, september 1973.

FLYNN, M.J. and ROSIN, R.F. "Microprogramming : an introduction and a viewpoint.
IEEE Trans computer, Vol. C-20, pp. 727-731, july 1971.

GLANTZ, H.T. "A note on microprogramming".
Journal of the ACM, Vol. 3 N° 2, p. 77, 1956.

GREEN. "Microprogramming emulators and programming languages".
Communication of the ACM, Vol. 9 n° 3, mars 1966.

GRIES, D. "Compiler construction for digital computers".
New-York, John Wiley and sons, Inc., 1971.

HILL, R.H. "Stored logic programming and applications".
Datamation, Vol. 10 N° 2, février 1964.

HUSSON, S.S. "Microprogramming : Principles and Practices".
Prentice Hall, Englewood Cliffs, N.J., 1970.

JACQUEMART, Y. "Interaction software-firmware".
Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique, Namur 1975.

LLOYD, G.R. et VAN DAM. "Design considerations for microprogramming language".
AFDIPS Conference Proceedings, Vol. 43, pp. 537-543, 1974.

Mc. GEE, W.C. "The TRW-13 computer".
Datamation, Vol. 10 N° 2, février 1964.

MELBOURNE et PUGMIRE. "A small computer for the direct processing of FORTRAN statements".
Computer Journal, Vol. 8 N° 1, p. 24-27, avril 1965.

MERCER, R.J. Microprogramming
Journal of the ACM, Vol. 4, p. 157, 1957.

OPLER. Fourth generation software, the realigment.
Datamation, Vol. 13 N° 1, pp. 22-24, janvier 1967.

RAMAMOORTHY, C.V. and TSUCHIYA, M. A high-level language for horizontal microprogramming.
IEEE transactions on computer, Vol. c-23, pp. 791-801, août 1974.

RAMAMOORTHY AND GONZALES, M.J. "A survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs".
AFIPS conference Proceedings, Vol. 35, pp. 1-15, 1969.

ROBERTS, P.S. and WALLACE, C.S. "A microprogrammed lexical processor".
Proceedings of the IFIP Congress, pp. 577-581, 1971.

ROSIN, R.F. "Contemporary concepts of microprogramming and emulation".
Computing surveys, Vol. 1, pp. 197-212, December 1969.

TESLER, L.G. and LENA, H.J. "A language design for concurrent Processes".
Spring Joint Computer Conference, Vol. 32, pp. 403-408, 1968.

TIRREL, A.K. "A study of the application of compiler techniques to the generation of microcode".
SIGMICRO/SIGPLAN interface meeting, may 1973.

TUCKER, S.G. "Emulation of large systems".
Communication of the ACM, Vol. 8 n° 12, pp. 753-761, december 1965.

VARIAN DATA MACHINES. "Varian 73, Processor Manual".
98A9906 021, mars 1973.

VARIAN DATA MACHINES. "Varian microprogramming Guide".
98A9906 072, august 1973.

WEBER, H. "A microprogrammed implementation of EULER on IBM 360 Model 30".
Communication of the ACM, Vol.10, pp. 549-558, september 1967.

WILKES, M.V. "The best way to design an automatic calculating machine".
Report of Manchester University Computer Inaugural Conference, p. 16, juli 1951.

WILKES, M.V. "The Growth of interest in microprogramming : a litterature survey".
Computing surveys, Vol. 1, pp. 139-145, september, 1969.

YAU, S.S., SCHOWE, A.C. and TSUCHIYA, M. "On storage Optimization for horizontal microprograms".
Micro 7, Preprints, pp. 98-105, 1974.

A N N E X E 1

Approche de la VARIAN

1. Généralités

    1.1. Description hardware
    1.1.1. Principe conducteur
    1.1.2. Description hardware
    1.2. Description de la micro-instruction

2. Interdépendance des champs

3. Description de l'adressage et expression des contraintes

    3.1. Adressage normal
    3.2. Adressage par un GOTO (IRy, MSKxxxxx, ...)
    3.3. Adressage par un GOTO (ad-vraie, ad-fausse)
    3.4. Saut de page
    3.5. Contrôle de routine
    3.6. Décodage des codes opérations

4. Résumé des contraintes à respecter

    4.1. de Hardware
    4.2. de Firmware
    4.3. de temps
    4.4. d'adressage

Annexe 1

Approche de la VARIAN

1. Généralités

1.1. Description hardware

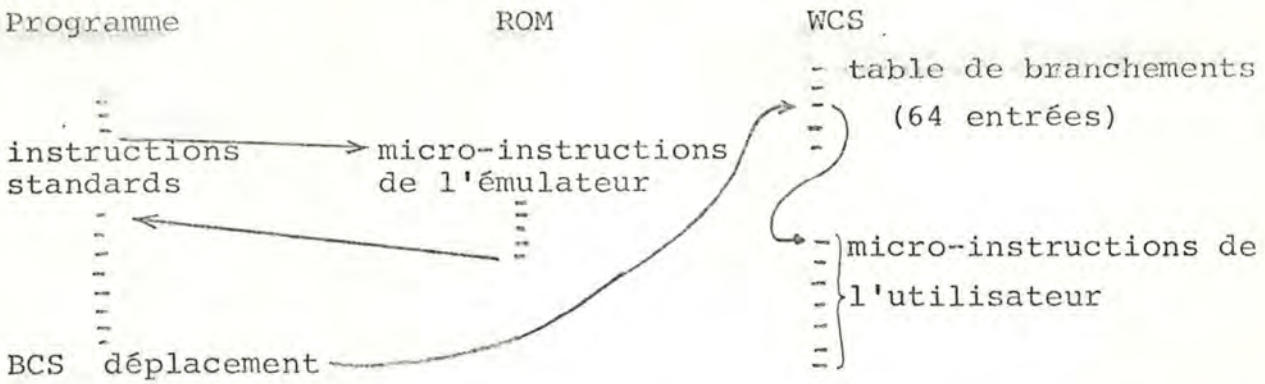
1.1.1. Principe conducteur

La série V70 de VARIAN a été conçue dans le but de remplacer le 620, micro-processeur destiné au contrôle et à la commutation de processus. Il fallait pouvoir remplir les rôles du 620 tout en permettant un accroissement des possibilités. La solution choisie a été la microprogrammation grâce à laquelle la 620 est émulée. Le développement de la série est rendu possible notamment par une W.C.S. Nous retrouverons dans la description hardware les deux grandes options prises : l'émulation de la 620 et la microprogrammation dynamique (fonction du temps et de l'utilisateur).

1.1.2. Description hardware

Partons du schéma du chemin des données (figure 2) et analysons brièvement les caractéristiques principales. D'abord, remarquons que si les 16 registres généraux de la microprogrammation sont connectés aux deux entrées de l'ALU, la mémoire ne l'est pas. Toute donnée venant de la mémoire ou allant vers la mémoire devra transiter dans un registre. Quelques registres spéciaux, sur lesquels certaines opérations peuvent s'effectuer sans l'interventionde l'ALU, apparaissent et font ressortir le souci de rapidité recherché. Ainsi le "Program counter" (P) peut s'incrémenter sans passer par l'ALU, le "shift counter" (SC) lui se décrémente indépendamment. L'"operand register" est un registre sur lequel les décalages peuvent s'effectuer en conjonction avec des opérations arithmétiques. Tous ces registres sont chargeables à partir de l'ALU. Du côté de la mémoire, un "pipe-line" (IBR-IR) pour une prise en charge rapide des instructions est créé. Deux micro-opérations de lecture en mémoire centrale sont donc créées (un "operand fetch "affectant le MIR et un "instruction fetch"affectant le MIR et l'IBR). L'adresse de la mémoire (MAD) peut venir du registre P de l'ALU ou du registre d'entrée mémoire (MIR). Signalons enfin qu'un masque peut être appliqué sur le registre instruction (IR).

La VARIAN est une machine à microprogrammation horizontale. Celle-ci est effectuée dans une mémoire inaltérable (512 mots de 64 bits) contenant l'ensemble standard des instructions de base de la 620. L'utilisateur peut y adjoindre une mémoire de microprogrammation altérable (WCS) pouvant atteindre 1.536 mots. Le contrôle est passé aux micro-routines de l'utilisateur par l'intermédiaire d'un BCS (branch to control store) qui n'est rien d'autre qu'un branchement vers une table de branchements de la WCS, celle-ci devant être chargée par l'utilisateur.



Dans la VARIAN, le contrôle des E/S et le décodage logique des instructions s'effectuent dans des ROM différentes séparées de celles contenant les micro-routines. Il est également possible de s'équiper de WCS distinctes pour ces parties. L'utilisateur peut ainsi non seulement ajouter de nouvelles primitives à l'ensemble des instructions, mais aussi modifier la logique standard du décodage (et donc le format des instructions) et la structure des E/S. Grâce à cela, nous pourrions créer une machine d'architecture complètement différente de celle définie dans les ROS (read only storage). Les problèmes des trois WCS peuvent être dissociés très facilement ce qui nous a permis de n'envisager dans le travail que le problème de l'élaboration d'un micro-assembleur pour les instructions traitant avec la mémoire.

1.2. Description de la micro-instruction

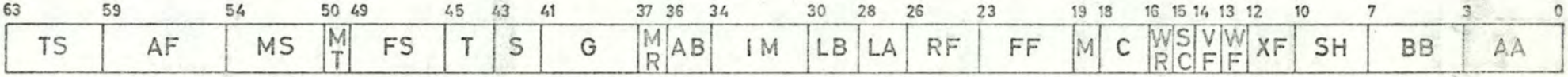
Elle se compose de 64 bits par lesquels le micro-programmeur contrôle le chemin des données. Le format de la micro-instruction comprend 25 champs ; le plupart d'entre-eux utilisent un encodage à deux ou même trois niveaux, c'est-à-dire que la signification d'un champ dépend de la valeur d'un autre champ de contrôle (voir fig. 1,(c)). Cette technique permet de réduire la longueur de la micro-instruction et donc la grandeur de la mémoire de contrôle,des conflits entre opérations de contrôle en découlent. Ainsi si on utilise une constante, certaines opérations de l'ALU sont inhibées. Les 25 champs indiquent plusieurs opérations : adressage de la mémoire de contrôle, test de conditions, sélection des données pour l'opération suivante, opérations mémoire, opérations arithmétiques et logiques. La figure 1 illustre le format de la micro-instruction et montre l'utilisation des différents champs.

2. Interdépendance des champs

L'analyse reprise à la figure 1 -b montre que l'interdépendance des champs est quasi nulle. Ce travail est un peu simpliste et cache tous les problèmes. Cette analyse ne fait pas ressortir l'encodage à différents niveaux dont nous avons parlé et qui est explicité pour un cas à la figure 1-c. Cette interdépendance de champs peut s'exprimer de différentes manières :

- par les tables séquentielles
- par un graphe
- par toute autre technique exprimant les dépendances.

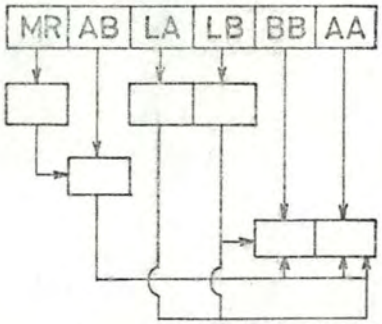
La figure 3 est un graphe global montrant au microprogrammeur



a) Schéma de la micro-instruction

Opérations	Champs utilisés
Formation de l'adresse	TS, AF, MS, MT, FS
Choix des fonctions de l'ALU	FF, MF, CF
Choix des opérandes	MR, AB, LA, LB, BB, AA
Choix de la destination	WR, RF, AA
Opérations avec la mémoire	SF, IM
Opérations de décalage	SH, XF, LA, LB, WF, SC
Opérations de test	GF, SF, TF

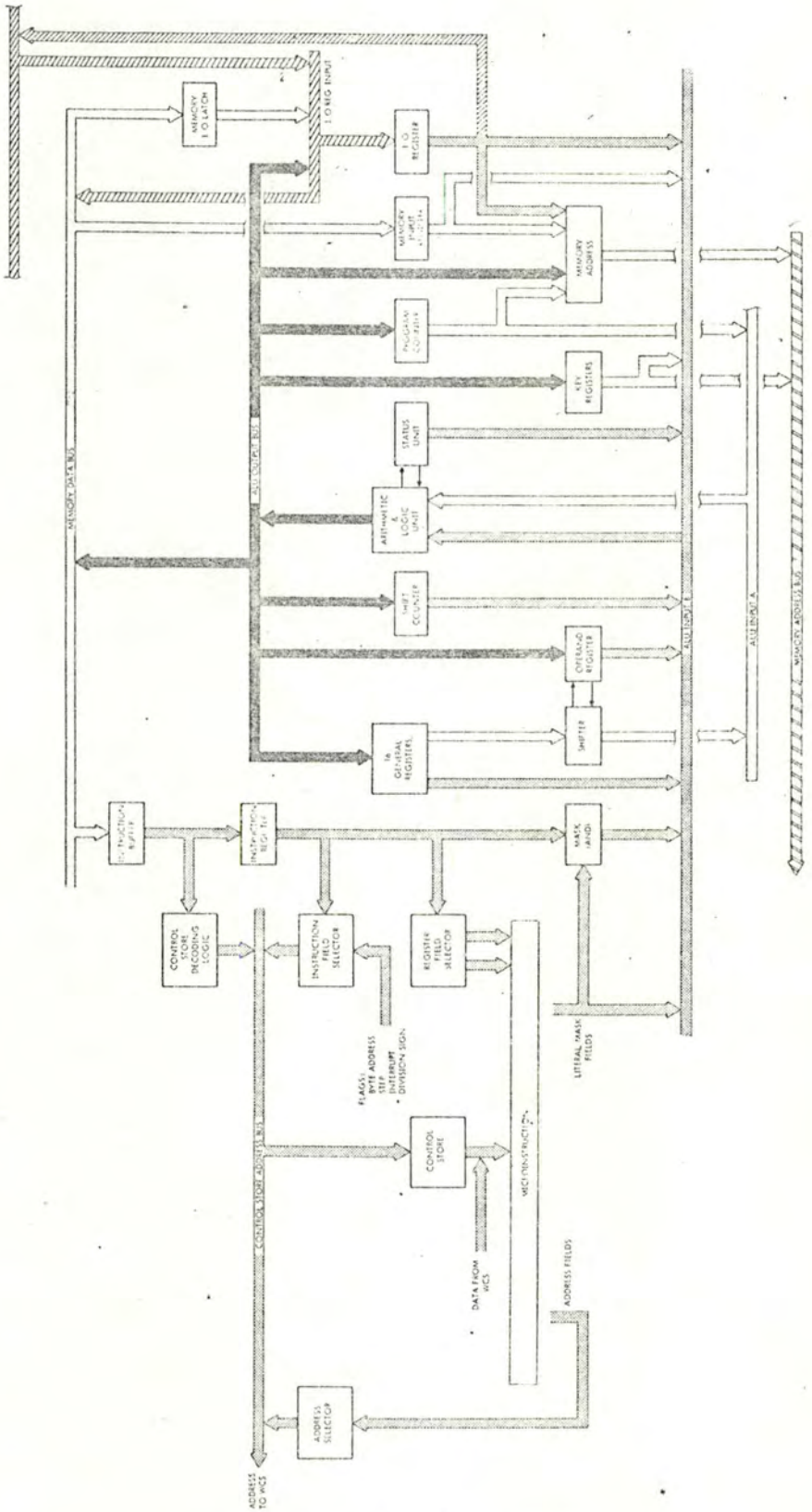
b) Utilisation générale des différents champs d'après les micro-opérations



Le décodage de MR conditionne la signification de AB, lequel oblitère éventuellement les champs LA et LB qui spécifiaient les opérandes de l'ALU.

c) Exemple d'encodage à trois niveaux

FIG. 1: Format de la micro-instruction



VT13-292

Figure 1-2. Varian 73 Processor Data Paths

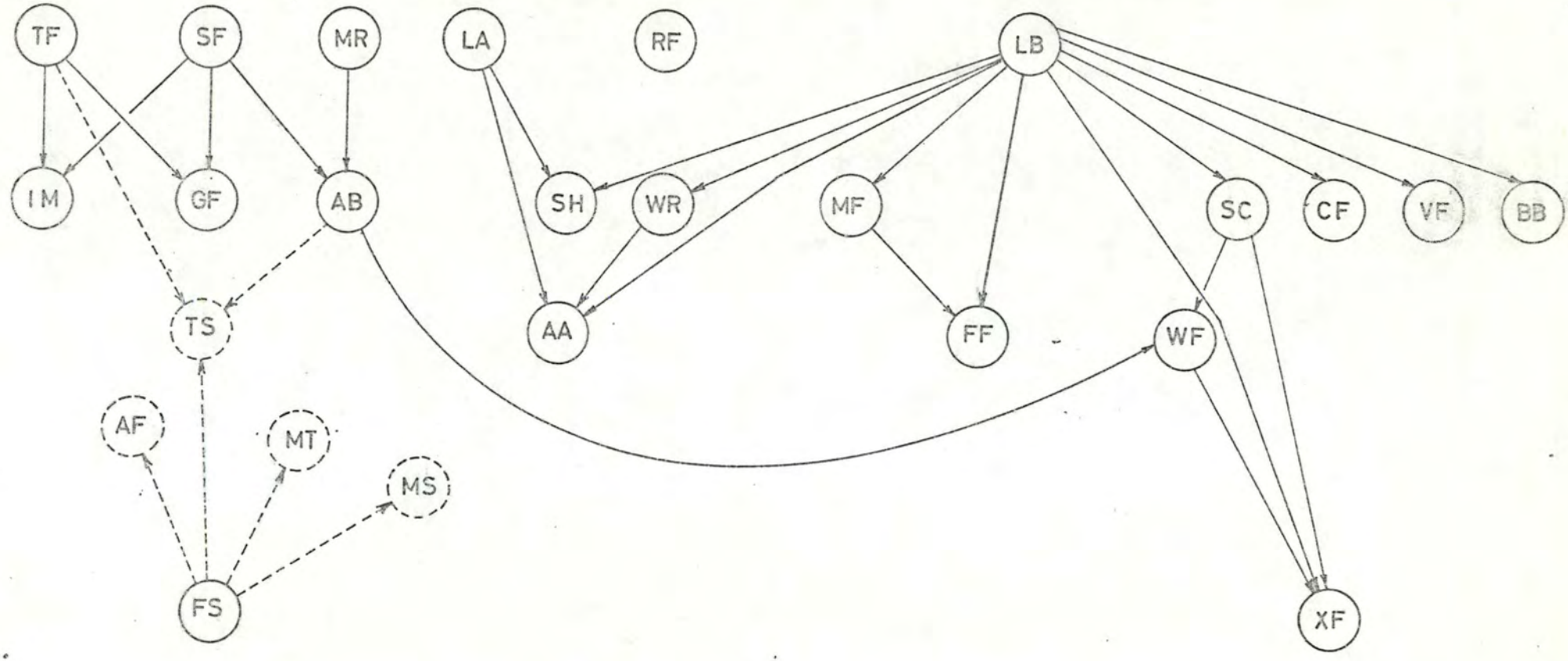


FIG. 3 : Dépendance générale des champs

les champs dont la valeur doit être contrôlée lors du positionnement de l'un d'entre-eux. Un automate devrait effectuer une analyse similaire, c'est-à-dire qu'il devrait balayer le graphe lors de chaque insertion potentielle de micro-opérations dans une micro-instruction. C'est une solution trop longue. Une table simplifierait déjà ce travail. Nous avons trouvé préférable de créer un langage intermédiaire reflètant la future micro-instruction, c'est-à-dire qu'il se compose des 64 bits de la micro-instruction. Il suffit dès lors de positionner les bits requis par la micro-opération et de comparer ces bits avec ceux de la micro-instruction au cours de la composition. Cette technique permet, sans perte de place, de gagner en nombre de comparaisons.

3. Description de l'adressage et expression des contraintes

Dans la microprogrammation de la VARIAN, un branchement n'est pas une opération séparée, mais il fait partie intégrante de la micro-instruction.

L'adresse suivante est déterminée soit par la micro-instruction en cours, soit par le décodeur. Dans le cas où l'adresse vient de la micro-instruction, elle est construite à partir de champs différents suivant qu'il s'agit d'un branchement inconditionnel ou d'un branchement conditionnel soit à un état soit au contenu du registre instruction (IR). Malgré ces différences, la formation de l'adresse s'organise toujours autour des champs AF, TS, FS, MT, MS combinés suivant le shéma de la figure 4 a. Les paragraphes suivants reprennent les différentes opérations de contrôle de séquence et ajustent le shéma général à chaque cas particulier.

3.1. Adressage normal (figure 4-b)

C'est l'adressage utilisé lorsque l'utilisateur spécifie un saut inconditionnel (JUMP (symb)). Aucune condition n'est donc spécifiée, aucune interruption n'est active et l'adressage par le décodeur n'est pas spécifié. Les champs FS-TS et MT sont mis à zéro de telle sorte que les 4 bits de poids faible sont donnés par le champ MS. Les cinq bits de poids fort sont donnés par le champ AF.

3.2. Adressage par un "GO TO (IRy, MSKxxxxx,ad-1, ... ad-i, ...)"

Un tel adressage utilise les bits y à y + 4 du registre instruction (IR) masqués (par MSK). Le paramètre IRy détermine la valeur du champ FS et le masque (MSK) se retrouve dans les champs MT et MS de la figure 4-a.

Dans le bas du graphique, se trouve le processus de sélection des bits du registre instruction (IR). $(FS)^{**}$ représente le contenu des bits de l'IR spécifiés par FS. Ainsi si FS = 4, $(FS)^{**}$ représente les bits 0 à 4 de l' IR. La partie supérieure forme les bits de l'adresse que l'on ne peut modifier par l'IR masqué. Une remarque (TS*) TS n'est pas utilisé pour former l'adresse lorsque : 1° on demande une extraction de registre (RSi);
2° on autorise les interrupts ;
3° lors d'un saut de page (JUMP (page, symb)
4° une opération de test est demandée (TEST ...).

Les valeurs 1, 2, 3 de FS serviront à coder les instructions :

GO TO (STEP, -, -) ⟶ FS = 3 et MT//MS = 2
GO TO (INT, -, -) ⟶ FS = 1 MT//MS = 1
GO TO (BYTA, -, -) ⟶ FS = 2 MT//MS = 1

3.3. Adressage par "GO TO (ad-vraie,ad-fausse)"

Ici, deux adresses doivent être spécifiées. L'adresse utilisée quand le test est vérifié est identique à celle formée par l'adressage ci-dessus (avec FS =0000). L'autre est formée par un "ou" entre les champs AF et TS d'après le schéma 4 c.

Cette description fait apparaître les contraintes que l'on devra respecter :

type JUMP ⟶ pas de contraintes
GO TO (IRy, MSK ...) ⟶ toutes les adresses doivent avoir cinq bits de poids fort identiques
GO TO (pass, fail) ⟶ idem + adresse non passante paire.

Les autres instructions de contrôle de séquences sont plus particulières et nécessitent généralement plus de bits.

3.4. Saut de page (JUMP (page, symb))

Le numéro de page est donné par le champ TS, d'où les limitations de son utilisation dans les trois grands types. Les 9 autres bits de l'adresse sont formés comme dans le cas de l'adressage normal.

3.5. Contrôle de routine (CALL, RETURN)

Ces instructions utilisent une pile dans laquelle le CALL place l'adresse de retour qui sera exploitée par le RETURN. Le CALL est en fait un JUMP (page, symb) avec sauvetage de l'adresse suivante sur la pile. Le numéro de page est donc placé dans le champ TS et les 9 bits restants sont établis comme dans l'adressage normal. L'adresse de retour comprend, elle aussi, treize bits (n° page + adresse normale). Celle-ci occupera les champs WR à BB comme indiqué dans la figure 4 d.

Lors de l'établissement des micro-opérations du langage intermédiaire, nous ne connaissons pas la valeur de ces dernières. Nous ne pouvons les mettre ni à zéro, ni à un, ni à la valeur quelconque, car ils risqueraient d'être utilisés lors d'une fusion.
Zéro étant représenté par 00, un par 01 et la valeur quelconque par 11 il reste le "10" pour signaler que ces bits seront positionnés ultérieurement.

3.6. DECODE

Nous devrions présenter ici par souci d'être complet la façon dont le décodeur travaille pour générer l'adresse. Il suffit de savoir qu'on l'utilise pour prendre en charge l'instruction suivante du langage machine et que le décodeur travaille sur l'IR qui doit contenir cette instruction. Pour la description complète nous renvoyons aux manuels du constructeur.

4. Résumé des contraintes à respecter

4.1. Hardware

+ Taille de la WCS ; 512 mots de 64 bits ;
+ certains status doivent être "échantillonnés" par firmware avant de pouvoir être testés ;
+ le chemin des données ne permet pas la banalisation des entrées dans l'unité arithmétique et logique ;
  - entrée A : Rn, Rn, SL ; Rn, SR ; P ; ZERO ; ONES ;
  - entrée B : Rn : MIR : IOR ; STAT ; OPR ; ORSE ; OLZF ; OFSF ; ORZF les "littéraux" et les masques MSKxxx ;
+ INCB et TCB exigent que l'entrée A soit ZERO ;
+ DECB exigent que l'entrée A soit ONES ;
+ dans le contrôle des champs AA et BB par analyse du registre instruction (IR), on ne peut contrôler qu'un seul champ tandis que l'autre reste à son ancienne valeur ;
+ positionnement du flag de retenue suivant fonction

4.2. Firmware

Les différents champs de la micro-instruction ne sont pas indépendants l'un de l'autre. Cette dépendance peut s'exprimer assez facilement par un graphe où chaque sommet représente un champ et chaque arc qui en part est nommé par une valeur prise par ce champ.

Dans le cas où l'opérande est un littéral ou MSKxxxx, les opérations INCB (incrémenter l'entrée B), SUB (substract), NOTB (inverser l'entrée B), TCB (two's complement) ne peuvent être effectuées.

4.3. Temps

+ Les micro-instructions suivant un OS ou un BS doivent conserver les données à sauver sur la sortie de l'ALU jusqu'à la fin de l'opération, c'est-à-dire jusqu'à la prochaine demande d'opération mémoire ou jusqu'au prochain WAIT ;
+ la plupart des micro-opérations sont terminées à la fin du cycle. Les exceptions sont :

+ l'incrémentation du "Pcounter" qui se passe à la moitié de la micro-instruction ;

  la valeur qui entre dans l'ALU est celle non incrémentée, par contre la nouvelle valeur sera prise pour adresse mémoire
+ le transfert de l'ALU vers l'instruction buffer register (IBR) et l'instruction register demande que la sortie de l'ALU soit maintenue pendant 2 micro-cycles.

4.4. Adressage

+ Dans l'adressage par sélection de champ GOTO (IRy, MSKxxxx,..) le champ TS n'est pas utilisé quand l'une des conditions ci-dessous est vérifiée ou si on utilise un TEST (TF ≠ 0).

  a) utilisation de la sélection de registre (AB = 01 V 10);
  b) les interrupt sont permis (S = T = 00 et GF = X1XX);
  c) un page-jump est spécifié (T = 00, S = 10, G = X1XX).

+ dans le cas d'une adresse par sélection de champs (GOTO (STEP, n_1, n_2), ... GOTO (IRx, MSK, n_1 ... n_n), l'adresse de base (AF, TS) doit être paire ;
+ dans l'adressage conditionnel (GOTO (vrai, faux), l'adresse de branchement dans le cas où le test donne une réponse négative doit : - être paire ;
  - être plus grande que l'adresse passante modulo 16, c'est-à-dire que les cinq bits de poids forts doivent être identiques.
+ dans l'adressage par sélection de champ, des adresses des micro-instructions à exécuter doivent être de la forme suivante :

	adresse passante	adresse non passante
GOTO (STEP, n_1, n_2)	xxxxxxxx1	xxxxxxxx0
GOTO (BVTA)	xxxxxxxx1	xxxxxxxx0
GOTO (SHFT)	xxxxxxx1X	xxxxxxx0X
GOTO (INT)	xxxxxxxx0	xxxxxxxx1